MEMORANDUM
RM-5883-PR
JANUARY 1969

AD 684124

DIGITAL COMPUTER SIMULATION: COMPUTER PROGRAMMING LANGUAGES

Philip J. Kiviat

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA · CALIFORNIA

110

MEMORANDUM
RM-5883-PR
JANUARY 1969

DIGITAL COMPUTER SIMULATION: COMPUTER PROGRAMMING LANGUAGES

Philip J. Kiviat

This research is supported by the United States Air Force under Project RAND—Contract No. F44620-67-C-0045—monitored by the Directorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of the United States Air Force.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA • 90406

PREFACE

This RAND Memorandum is one in a continuing series on the techniques of digital computer simulation. Each Memorandum covers a selected topic or subject area in considerable detail. This study discusses computer simulation programming languages. It describes their characteristics, considers reasons for using them, compares their advantages and disadvantages relative to other kinds of programming languages and, through examples, compares four of the most popular simulation languages in use today.

The Memoranda are being written so that they build upon one another and provide an integrated coverage of all aspects of simulation. The only Memorandum in the series that needs to be read before this one is P. J. Kiviat, Digital Computer Simulation: Modeling Concepts, The RAND Corporation, RM-5378-PR, August 1967. All the Memoranda should be of particular interest to personnel of the AFLC Advanced Logistics System Center, Wright-Patterson Air Force Base, and to Air Force systems analysts and computer programmers. Persons responsible for selecting simulation programming languages for particular projects or for installations of computer systems should find this Memorandum particularly useful.

SUMMARY

Simulation programming languages are designed to assist analysts in the design, programming, and analysis of simulation models. This Memorandum discusses basic simulation concepts, presents arguments for the use of simulation languages, discusses the four languages GPSS, SIMSCRIPT II, SIMULA, and CSL, summarizes their basic features, and comments on the probable future course of events in simulation language research and development.

Simulation languages are shown to assist in the design of simulation models through their "world view," to expedite computer programming through their special purpose, high-level statements, and to encourage proper model analysis through their data collection, analysis, and reporting features. Ten particularly important simulation programming language features are identified: modeling a system's static state, modeling system dynamics, statistical sampling, data collection, analysis and display, monitoring and debugging, initialization and language usability. Examples of each of the four simulation languages, GPSS, SIMSCRIPT II, SIMULA, and CSL, are used to illustrate how these features are implemented in different languages.

The future development of simulation programming languages is shown to be dependent on advances in the fields of computer languages, computer graphics, and time sharing. Some current research is noted; outstanding research areas are identified.

CONTENTS

I. INTRODUCTION

The introductory Memorandum in this series presented a rationale for simulation, discussed why simulation experiments are performed, and pointed out that, while computers are not mandatory for simulation, most simulations today require computers because of their complexity and sampling requirements [34]. Few aspects of computer technology are vital to simulation,* since one can perform simulations without specialized equipment. Computers make it easier to perform a simulation study, however, and the frequent savings in time and expense allow more time to be spent determining the reliability of simulated results and designing simulation experiments.

Specialized computer simulation equipment can take the form of either hardware (computers and peripheral equipment) or software (compilers, assemblers, operating systems). This Memorandum is dedicated to software. It discusses simulation languages, describes their characteristics, considers reasons for using them, and compares their advantages and disadvantages relative to other kinds of programming languages.

SOME DEFINITIONS

A reader completely unfamiliar with digital computers and the basic concepts of computer programming should consult an introductory computer programming text before going any further. References [1] and [22] are good texts for the purpose. Readers familiar with computers and at least aware of the basic concepts of programming should be able to follow this Memorandum without additional preparation.

A computer programming language is a set of symbols recognizable by a computer, or by a computer program, that denote operations a programmer wishes a computer to perform. At the lowest level, a basic machine language (BML) program is a string of symbols that corresponds directly to machine functions, such as adding two numbers, storing a number, and transferring to an address. At a higher level, an assembly

*Excluding analog and hybrid simulation, of course.

language (AL) program is a string of mnemonic symbols that correspond to machine language functions and are translatable into a basic machine language program by an assembly program or assembler. Simple assemblers do little but substitute basic machine language codes for mnemonics and assign computer addresses to variable names and labels. Sophisticated assemblers can recognize additional symbols (macros) and construct complicated basic machine language programs from them.

A compiler is a program that accepts statements written in a usually complex, high-level compiler language (CL) and translates them into either assembly language or basic machine language programs -- which may in turn, at least in the case of CL to AL translation, be reduced to more basic programs. Compilation is much more complex than assembly, as it involves a higher level of understanding of program organization, much richer input languages, and semantic as well as syntactic analysis and processing.

An interpreter is a program that accepts input symbols and, rather than translate them into computer instructions for subsequent processing, directly executes the operations they denote. For this reason, an interpretive language (IL) can look like a BML, an AL, a CL or anything else. Interpretive language symbols are not commands to construct a program to do something, as are assembly language and compiler language commands, but commands to do the thing itself. Consequently, even though programs written in a CL and an IL may look identical, they call for sharply different actions by the programs that "understand" them, and different techniques are employed in writing them.

For all but basic machine language and interpretive programs, a distinction has to be drawn between the program submitted to the computer, the source language program, and the program executed* by the computer, the object program. An assembler that accepts mnemonic basic machine codes as its input and translates them into numerical basic machine codes has the mnemonics as its source language and the numerical basic codes as its object language. A compiler that accepts English-like language statements as its input and translates them into assembly

*Excluding modifications made during loading.

language mnemonics, which are in turn translated into numerical basic machine codes, has the English-like language as its source language and the numerical basic codes as its object language. An interpreter that operates by reading, interpreting, and operating directly on source codes has no object code. Every time an interpretive program is executed, a translation takes place. This differs from what is done by assemblers and compilers where translation takes place only once, from source to object language, and thus enables the subsequent running of object programs without translation.

Basic machine language and assembly language programs suffer in that they are specific to a particular computer. Since their symbols correspond to particular computer operations, programs written in BML or an AL are meaningful only in the computer they are designed for. As such, they can be regarded as <u>machine oriented languages</u> (MOL).

Most compilers and interpreters can be classified as <u>problem oriented languages</u> (POL). As such, they differ from BML and AL that reflect computer hardware functions and have no problem orientation. A POL written for a particular problem area contains symbols (language statements) appropriate for formulating solutions to typical problems in that area. A POL is able to express problem solutions in <u>computer independent notation</u>, using a program that "understands" the POL to translate the problem solution expressed in source language to a BML object program or execute it interpretively.

Figure 1 illustrates a BML, an AL, and two POLs. Each example shows the statement or statements (symbols) that must be written to express the same programming operation, the addition of three numbers.

BML	AL: FAP	POL: FORTRAN	POL: COBOL
+050000 ...	CLA A	X=A+B+C	ADD A,B TO C GIVING X
+040000 ...	ADD B		
+040000 ...	ADD C		
+060100 ...	STO X		

Fig. 1 -- A programming example

The point of the discussion so far has been to establish the definitions of BML, AL, CL, MOL, POL, assembler, compiler, and interpreter. Without these definitions it is impossible to understand the historical evolution of simulation programming languages or their basic characteristics.

A simulation programming language (SPL) is a POL with special features. Simulation being a problem-solving activity with its own needs, programming languages have been written to make special features available to simulation programmers at a POL level. Historically, this has been an evolutionary process. SPLs have developed gradually from AL programs with special features, through extended commercially available POLs, to sophisticated, special-purpose SPLs. Some discussion of these special features is necessary to place the development process in perspective and introduce the topics that follow. More complete histories of simulation programming languages and the development of simulation concepts can be found in Refs. 35, 36, 42, 43, 46, 60, and 61.

PRINCIPAL FEATURES OF SIMULATION LANGUAGES

Simulation, as defined in [34], is a technique used for reproducing the dynamic behavior of a system as it operates in time.

To represent and reproduce system behavior, features not normally found or adequately emphasized in most programming languages are needed. These features:

(1) Provide data representations that permit straightforward and efficient modeling,

(2) Permit the facile portrayal and reproduction of dynamics within a modeled system, and

(3) Are oriented to the study of stochastic systems, i.e., contain procedures for the generation and analysis of random variables and time series.

The first of these features calls for data structures more elaborate than the typical unsubscripted-subscripted variable organizations found in, say, FORTRAN and ALGOL. Data structures must be richer in two ways: they must be capable of complex organization, as in tree structures, lists, and sets; and they must be able to store varieties of data, such as numbers, both integer and real, double-precision and complex, character strings of both fixed and variable length, and data structure references. As data structures exist only so that they can be manipulated, statements must be available that (1) assist in initializing a system data base (as we may call the collection of data that describe a system); (2) permit manipulations on the data, such as adding elements, changing data values, altering data structures and monitoring data flows; and (3) enable communication between the modeler and the data. PL/I, the newest general-purpose POL, pays great attention to data structures, although not as much as some people would like [29]. ALGOL 68, the revised version of ALGOL 60, also leans in this direction [18]. Activity in the CODASYL committee charged with improving COBOL shows that they too are aware of the importance of this topic [55].

The second of the features deals with modeling formalisms, both definitional and executable, that permit the simulation of dynamic, interactive systems. Statements that deal with time-dependent descriptions of changes in system state, and mechanisms that organize the execution of various system-state-change programs so that the dynamics of a system are represented correctly, are an integral part of every SPL.

The third of the features stems from the fact that the world is constantly changing in a stochastic manner. Things do not happen regularly and deterministically, but randomly and with variation. Procedures are needed that generate so-called pseudorandom variates from different statistical distributions and from empirical sampling distributions, so that real-world variability can be represented. Procedures are also needed for processing data generated by simulation models in order to make sense out of the masses of statistical data they produce. [21]

The history of simulation-oriented programming languages noted above points out that there is no one form a simulation language must take, nor any one accepted method of implementing such a language. An SPL can be an AL with special instructions in the form of macros that perform simulation-oriented tasks, a CL with special statements that perform essentially the same tasks, or an IL with statements similar to those found in simulation-oriented CLs and ALs but with an entirely different implementation. It is sufficient here merely to point out the principal characteristics of all SPLs, providing a base for discussing why SPLs are needed and for understanding some pros and cons of using specialized SPLs and general POLs for simulation. Sections II and III discuss the concepts of discrete-event simulation in some detail and thoroughly explore the features noted above.

REASONS FOR HAVING SPLs

The two most frequently cited reasons for having simulation programming languages are (1) programming convenience and (2) concept articulation. The former is important in the actual writing of computer programs, the latter in the modeling phase and in the overall approach taken to system experimentation.

It is difficult to say which of the two is more important. Certainly, many simulation projects have never gotten off the ground, or at least were not completed on time, because of programming difficulties. But then, other projects have failed because their models were poorly conceived and designed, making the programming difficult and the required experimentation impossible or nearly so. If it were necessary to choose, concept articulation should probably be ranked first, since any statements or features provided by a simulation programming language must exist within a conceptual framework.

Succeeding sections examine a number of simulation programming concepts and how they are implemented in different SPLs. Some models are also described, with comments on how various conceptual frameworks help or hinder their analysis and examination.

It is fair to say at this point, before going through this demonstration and without documented proof, that SPLs have contributed to the success of simulation as an experimental technique, and that the two features, programming convenience and concept articulation, are the major reasons for this success. SPLs provide languages for describing and modeling systems, languages composed of concepts central to simulation. Before these concepts were articulated, there were no words with which to describe simulation tasks, and without words there was no communication -- at least no communication of the intensity and scope to be found today.

A third substantial reason for having higher-level SPLs has come about through their use as communication and documentation devices. When written in English-like languages, simulations can be explained to project managers and nonprogramming-oriented users much more easily than when they are written in hieroglyphic ALs. Explanation and debugging go easier when a program can be read rather than deciphered.

REASONS FOR USING EXISTING POLs

Cogent arguments, both technical and operational, have been advanced for avoiding SPLs and sticking with tried-and-true algebraic compilers. Technical objections dwell mostly on object program efficiency, debugging facilities, and the like. Some of the operational objections are the noted inadequacy of SPL documentation, the lack of transferability of SPL programs across different computers, and the difficulty of correcting SPL compiler errors.

Most of these points are valid, although their edge of truth is often exceedingly thin. It is almost necessarily true that specialized simulation programming languages are less efficient in certain aspects than more general algebraic compilers. Because an SPL is designed for one purpose, it is less efficient for another. No single programming language can be all things to all men, at least not today. Painful experience is proving this to be true. SPLs should be used where their advantages outweigh their disadvantages, but not criticized for their limitations alone. An SPL should be criticized if it does something poorly it was designed to do, i.e., a simulation-oriented task, but not if it is inefficient in a peripheral nonsimulation-oriented task.

But technical criticisms are the least of the arguments levied against SPLs by people seeking to justify their use of existing algebraic POLs. The most serious and justifiable criticisms are those pertaining to the use of individual SPLs. Unlike the commonly used POLs, such as FORTRAN, ALGOL, and COBOL, which are produced and maintained by computer manufacturers, SPLs, with few exceptions, have been produced by individual organizations for their own purposes and released to the public more as a convenience and intellectual gesture than a profitable business venture. The latter are too often poorly documented, larded with undiscovered errors, and set up to operate on nonstandard systems, or at least on systems different from those a typical user seems to have. While attractive intellectually, they have often been rejected because it is simply too much trouble to get them working.* In a programming community accustomed to having computer manufacturers do all the compiler support work, most companies are not set up to do these things themselves.

The answer has been, "Stick to FORTRAN or something similar." It is easy to sympathize with this attitude, but it is unwise to agree in all cases. For a small organization with limited programming resources, doing a small amount of simulation work under such a strategy is probably justifiable; difficulties can be eased somewhat by using languages such as GASP and FORSIM IV that are actually FORTRAN programming packages [19], [37], [53]. Large organizations that have adequate programming resources and do a considerable amount of simulation work are probably fooling themselves when they avoid investing resources in an SPL and stick to a standard POL. One reason they often decide to do so is that the direct costs to install and maintain a SPL are visible, while the incremental costs incurred by using a POL are hidden and not easily calculated. This is the worst kind of false economy. Another often-heard excuse is that programmers and analysts are unwilling to learn a new programming language. If so, they should reform. When they learn to use an SPL, they are doing far more than learning a new programming language; they are learning concepts especially structured for simulation modeling and programming -- concepts that do not even exist in nonsimulation-oriented POLs.

Today, the designers of simulation programming languages are paying much more attention to their users than they have in the past, and computer manufacturers are supporting SPLs much more readily. While the era of the independently produced SPL is not past, it has probably seen its heyday. Problems of system compatibility and compiler support

*With the exception of GPSS, which IBM introduced and has maintained, supported, and redesigned three times since 1962.

will diminish in the future, and most operational problems will fade or vanish. But there is no escaping the need to learn new languages; our only choice is whether to volunteer or be drafted.

II. SIMULATION PROGRAMMING CONCEPTS

Every SPL has a small number of special simulation-oriented features. The way they are elaborated and implemented makes particular SPLs difficult or easy to use, programmer- or analyst-oriented, etc. They support the concepts embodied in the definition of simulation used in this series of Memoranda: the use of a numerical model to study the behavior of a system as it operates over time.

Taking the key words in this definition one at a time sets forth basic SPL requirements:

Use . . . to study the behavior: an SPL must provide facilities for performing experiments, for presenting experimental results, for prescribing experimental conditions, etc.

Numerical model . . . of a system: an SPL must provide facilities for describing the structure of a great variety of systems. Representations are needed for describing the objects found in systems, their qualities and properties, and relationships between them.

Operates over time: an SPL must provide facilities for describing dynamic relationships within systems and for operating upon the system representation in such a way that the dynamic aspects of system behavior are reproduced.

This section concentrates first on concepts related to descriptions of a system's static structure and next on concepts related to representing system dynamics. Section III discusses features needed for the efficient and practical use of simulation models.

DESCRIBING A SYSTEM: THE STATIC STRUCTURE

The static structure of a simulation model is a time-independent framework within which system states are defined. System states are possible configurations a system can be in; in numerical models, different system states are represented by different data patterns. Dynamic system processes act and interact within a static data structure, changing data values and thereby changing system states.

A definition of a system points out characteristics that are important in establishing a static system structure: a system is an interacting collection of objects in a closed environment, the boundaries of which are clearly stated. Every system:

(a) contains identifiable classes of objects,

(b) which can vary in number,

(c) have varying numbers of identifying characteristics,

(d) and are related to one another and to the environment in changeable, although prescribed ways.

Simulation programming languages must be able to:

(1) define the classes of objects within a system,

(2) adjust the number of these objects as conditions within the system vary,

(3) define characteristics or properties that can both describe and differentiate objects of the same class, and declare (numerical) codes for them, and

(4) relate objects to one another and to their common environment.

These requirements are not unique to SPLs; they are also found in languages and programs associated with information retrieval and management information systems.

While it might be interesting to examine all SPLs and contrast the particular ways in which they express structural concepts, it would hardly be practical. For one thing, they are too numerous; for another, many are simply dialects, lineal descendants or near relatives of a small number of seminal languages. In the interests of economy and clarity, only the basic concepts of these languages are discussed here. Excellent discussions of the features and pros and cons of the most widely used simulation languages can be found in Refs. 43, 60, 61, 64, and 66.

Identification of Objects and Object Characteristics

All SPLs view the "real world" in pretty much the same way, and reflect this view in the data structures they provide for representing systems. Basically, systems are composed of classes of different kinds of objects that are unique and can be identified by distinguishing characteristics. Objects are referred to by such names as entity, object, transaction, resource, facility, storage, variable, machine, equipment, process, and element. Object characteristics are referred to by such names as attribute, parameter, state, and descriptor. In some languages all objects are passive, i.e., things happen to them; in some languages objects are active as well, i.e., they flow through a system and initiate actions.

Table 1 lists several popular SPLs and shows the concept names and formalisms associated with each.

Table 1

IDENTIFICATION METHODS

Language	Concepts	Example	
SIMSCRIPT [33, 39, 45]	Entity, Attribute	AGE(MAN)	read AGE OF MAN
SIMULA [13, 14, 59]	Process, Attribute	AGE	attribute of current process MAN
GPSS [23, 24, 26]	Transaction, Parameter	P1	first parameter of current transaction
CSL [7, 9, 11]	Entity, Property	LOAD(SHIP)	read LOAD OF SHIP

Relationships between Objects

There is a class relationship between objects in all SPLs; several objects have different distinguishing characteristics and are in that sense unique, but have a common bond in being of the same type. For example, in a system containing a class of objects of the type SHIP, two ships may have the names MARY and ELISABF"" The objects are different yet related.

This form of relationship is rarely strong enough for all purposes, and must be supplemented. It is almost always necessary to be able to relate objects, of the same and different classes, having restricted physical or logical relations in common. For example, it might be necessary to identify all SHIPs of a particular tonnage or all SHIPs berthed in a particular port.

To this end, all SPLs define relationship mechanisms. Names such as set, queue, list, chain, group, file, and storage are used to describe them. Each language has operators of varying power that place objects in, and remove them from relationship structures, determine whether several objects are in particular relationships to each other, and so on.

Table 2 lists the relationship concepts of the languages shown in Table 1.

Table 2

RELATIONSHIP METHODS

Language	Concept	Example
SIMSCRIPT	Set	FILE MAN FIRST IN SET(I); insert MAN into SET(I)
SIMULA	Set	PRCD(X,MAN); precede element X with element MAN in the set to which X belongs
GPSS	User chain group	LINK I, FIFO; put current transaction first in Chain I
CSL	Set	MAN.3 HEAD SET(I); put the third man at the head of Set I

<u>Generation of Objects</u>

Some languages deal only with fixed data structures that are allocated either during compilation or at the start of execution. These structures represent fixed numbers of objects of different classes. Other languages allow both fixed and varying numbers of objects. There is a great deal of variety in the way different languages handle the generation of objects. The methods are related both to the "world view" of the language and the way in which the language is expressed, i.e., as a compiler, an interpreter, or a POL program package. Many of the differences between SIMSCRIPT and SIMULA can be traced to compiler features that have little to do with simulation per se. The block-structure/procedure orientation of SIMULA, which is rooted in ALGOL, has influenced the way processes are generated and the way they communicate with one another. The global-variable/local-variable/subroutine orientation of SIMSCRIPT, which is rooted in FORTRAN, has similarly influenced the way entities are generated and the way they communicate with one another. In these two cases, the differences are profound. A SIMULA process contains both a data structure and an activity program; a SIMSCRIPT entity holds only a data structure and is linked indirectly to an event subroutine. Some of the consequences of this division can be seen in the examples of Sec. IV.

Table 3 describes several object generation methods.

Table 3

GENERATION METHODS

Language	Concept	Example
SIMSCRIPT	Generate a new entity whenever one is needed	CREATE A MAN CALLED HENRY
SIMULA	Generate a new process whenever one is needed	HENRY:= <u>new</u> MAN
GPSS	Generate a new transaction with some specified time between successive generations	GENERATE 10,3
CSL	Does not exist	--

Of necessity, these illustrations are sketchy and not indicative of the wealth of descriptive, relational and operational facilities offered by the languages quoted. This is not altogether bad, as the purpose here is to impart a flavor for the ways in which SPLs describe static system structures and not to teach or compare features of particular languages. The reader who is interested in the specifics of individual languages should refer to their respective programming manuals.

DESCRIBING A SYSTEM: THE DYNAMIC STRUCTURE

While a model's static structure sets the stage for simulation, it is its dynamic structure that makes it possible. The dynamics of system behavior in all SPLs is represented procedurally, that is, by computer programs. While desirable, no nonprocedural SPLs have yet been invented, although substantial success toward this end has been achieved in limited areas [25].

At present, two SPLs have achieved widespread prominence and use in the United States, and two others have achieved similar prominence in Europe and Great Britain. These are GPSS, SIMSCRIPT, SIMULA, and CSL, respectively. Interestingly enough, each presents a different view of system dynamics. To understand why this is so, a historical rather than functional discussion seems appropriate.

The Concept of Simulated Time

Soon after academics and practitioners recognized that simulations of industrial and military processes could be conducted on digital computers, they started to separate the simulation-oriented portions of computer programs from the parts describing the processes being simulated. A simulation vocabulary was developed; the first word in it was probably "clock." Program structures began to reflect the concepts embodied in the vocabulary.

Since time and its representation are the essence of simulation, it was natural for it to be the first item of concern. If one could represent the passage of time within a computer program and associate the execution of programs with specific points in this simulated time, one could claim to have a time-dependent simulation program.

The first simulation clocks imitated the behavior of real ones. They were counters that "ticked" in unit increments representing seconds, minutes, hours, or days, providing a pulse for simulation programs. Each time the clock ticked, a simulation control program looked around to see what could happen at that instant in simulated time. What could happen could be determined in two ways: by predete [illegible] truction or by search. Before going into these two techniques, [illegible] me words are in order about simulation control programs.

The Structure of Simulation Control Programs

The heart of every simulation program, and every SPL, is a time control program. This program is referred to in various publications as a clockworks, a simulation executive, a timing mechanism, a sequencing set, and the like. Its functions are always the same: to advance simulation time and to select a subprogram for execution that performs a specified simulation activity.

Thus, every simulation program has a hierarchical structure. At the top sits the time control program, at an intermediate level sit simulation-oriented routines, at the bottom sit routines that do basic housekeeping functions such as input, output, and the computation of mathematical functions. Every SPL provides a time control program; when using an SPL, a simulation programmer does not have to write one himself -- or even worse, invent one.

Depending on how the time control program works, a simulation programmer may or may not have to use special statements to interact with the timing mechanism. Most simulation languages contain one or more statements that permit a programmer to organize system activities in a time-dependent manner. Further on, this section describes several different simulation control program schemes and the ways in which a programmer interacts with them.

First, it must be understood that every simulation program is composed of blocks of statements that deal with specific system activities. These blocks may be complete routines* or parts of routines. They have been called events, activities, blocks, processes, and segments. The distinctions between them will be clarified presently; at the moment it is only necessary to understand that a simulation program is composed of identifiable modules that deal with different simulation situations.

A simulation control program can select a portion of code to execute in either of two ways: by predetermined instruction or by

*The words routine, subroutine, program, subprogram, and procedure are used here interchangeably.

search. Regardless of how the result is determined, the effect is the same -- the execution of an appropriate block of code. Figure 2 blocks out the basic structure of every simulation program.

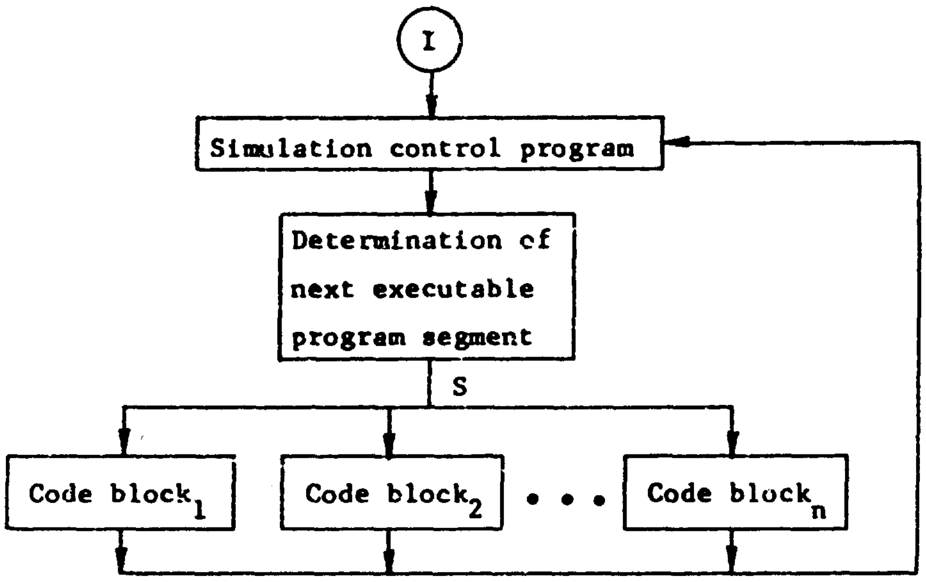

I
Simulation control program
Determination of next executable program segment
S
Code block$_1$
Code block$_2$
• • •
Code block$_n$

Fig. 2 -- Basic simulation structure

Simulation starts at I, where a model is initialized with sufficient data to describe its initial system state and the processes that are in motion within it. Based on information computed in the "next event" block, S switches to the code block that corresponds to the proper simulation activity.

The "search" method of next-event selection relies upon the fact that when a system operates it moves from state to state in a predetermined manner. The times at which state changes occur may be random, and represent the effects of statistically varying situations, but basic cause-and-effect relations still hold. Given that a system is in state "A", it will always move into state "B" if certain conditions hold; code block AB, say, must always be executed to effect the change. A "search" method relies upon descriptions of activity-producing system states and a scanner that examines system state data to determine whether a state change can take place at any particular clock pulse.

When a state change can be made, the code block representing it alters data values to reflect the change. Since many system changes take place over a period of time, some of the data changes are to "entity clocks." These clocks are set to the simulated time at which a state change is considered completed. When the control program finds that an "entity clock" has the same time as the master simulation clock, it performs the activity associated with that clock, e.g., relegating a working machine to idleness or causing an emptying tank to run dry. State changes that happen instantaneously, either when a code block is executed or as the result of some entity clocks equaling the simulation clock, cause new code blocks to be executed, new entity clocks to be set, . . ., cause the system activities to be reexamined.

The efficiency of this rather basic scheme was first improved by eliminating the uniform clock pulse. Since, in many simulations, events do not occur on every clock pulse but randomly in time, a great deal of computer time can be lost in scanning for things to do each time the clock is advanced one increment. It is more efficient to specify the time at which the <u>next event</u> is to occur and to advance the clock to this time. As nothing can happen before this time, it is unnecessary to search for altered system states. By definition of the next event, no entity clock can have an earlier time. At best, it can only be equal to the next event time.

The term "next-event simulation" was given to simulation programs that stepped from event time to event time, passing over increments of time in which no state changes occurred. <u>All modern SPLs use the next-event technique</u>. The term <u>critical event</u> is often used in the same spirit.

Two SPLs that do employ search are GSP [62] and CSL. In both, the <u>activity</u> is the basic dynamic unit. An activity is a program composed of a test section and an action section. Whenever simulation time is advanced, all activity programs are scanned for possible performance. If all test conditions in an activity are met, state-changing and time-setting instructions in the action section are executed; if at least one test condition is not met, the action instructions are passed over.

A cyclic scanning of activity programs insures that all possibilities are examined and all interactions are accounted for.

In addition to the activities scan, GSP incorporates an event scheduling mechanism that enables an activity to specify that some system event is to take place at a determined time in the future. Events that are not affected by other events, i.e., are not heavily interactive, can be treated more efficiently this way, as repeated scanning is not required to determine when they can be done.

When an activity scan is not employed, as is the case in GPSS, SIMSCRIPT, and SIMULA, all system events must be predetermined and scheduled. The activity-scan and event-scheduling approaches are different solutions to the same problem; an activity scan is efficient for highly interactive processes involving a fixed number of entities, e.g., multiresource assignment problems in shops producing homogeneous products; event scheduling is efficient for less interactive processes involving large numbers of entities, e.g., simulations of job shops producing special order products. Efficiency must be treated as a multidimensional quality, of course. We must speak of modeling efficiency and programming efficiency, as well as computer running-time efficiency.

The differences between activity scanning and event scheduling orientations can be pointed out best by procedural descriptions.

Event Selection Procedures

Take a simple shop situation in which a man and a machine must work together to produce a part. Each has an independent behavior, in that the man starts his day and ends it, takes coffee breaks, and goes to lunch without regard for how the machine is performing, and the machine suffers breakdowns and power failures without regard for what the man may be doing.

The Activity Scanning Approach. An activity approach to simulating the processing of a part in this man-machine shop specifies the conditions for a job to start processing, and the actions that take place when such conditions are met:

```
Test section:       if part is available AND
                    if machine is idle   AND
                    if man is idle       THEN do Action section
                    OTHERWISE return to timing mechanism

Action section:     put man in committed state
                    put machine in committed state
                    determine time man will be engaged
                    determine time machine will be engaged
                    set man-clock to time man will become
                         available
                    set machine-clock to time machine will
                         become available
                    return to timing mechanism
```

Emphasis is on the activity of producing the part, not on the individual roles of the man and the machine. Periodic scanning of the activity finds instances when all three conditions hold.

The Event Scheduling Approach. An event scheduling approach to the same problem requires that three programs be written, one for the man, one for the machine, and one for the part. The programs contain both test and action statements, and are "menus" for situations that can take place whenever a state change event occurs. For example, one event in the simulation of the above man-machine shop would be the return of a man to the idle state from whatever activity he might have been engaged in. The routine that represents the "man becomes idle" event might look like:

Test section:	if part is available AND if machine is idle THEN do Action section$_1$ OTHERWISE do Action section$_2$
Action section$_1$:	put man in committed state put machine in committed state determine time man will be engaged determine time machine will be engaged schedule return of man to availability schedule return of machine to availability return to timing mechanism
Action section$_2$:	put man in idle state return to timing mechanism

While these two program protocols may look similar, they are quite different. The event program is executed only when a state change occurs; the activity program, on the other hand, is examined at each timing cycle to see if a state change can take place. Furthermore, the activity contains logic for the availability of part, man, and machine, while three event programs must be written for the return of a man, machine, and part -- testing, respectively, that a machine and part, man and part, and man and machine are available.

Neither approach is clearly superior to the other; each has its advantages in some situations. Differences among SPLs that utilize one approach or another usually stem from their authors' attempts to design a language suited to the particular class of problems they study and hence gain modeling, programming, and execution efficiency.

The Process Interaction Approach. One of the difficulties of the event approach is its division of the logic of an operating system into small parts. The activity approach seems to suffer less from this criticism. A third approach, called the process, attempts to combine the efficiencies of event scheduling with the concise notation of activity scanning.

A process can be defined as a set of events that are associated with a system behavior description. The events are interrelated by special scheduling statements, such as DELAY, WAIT, and WAIT UNTIL, that interrupt the execution of a subprogram until a specified period of time has passed or a stated set of conditions hold. DELAY and WAIT are time-oriented and are effected through event scheduling techniques. WAIT UNTIL, being condition-oriented, requires an activity-scan approach. A process description thereby combines the run-time efficiency of event scheduling with the modeling efficiency of activity scanning. SIMULA is a process-oriented language that has had several years of successful experience and has undergone one revision [16]. GPSS is a process-oriented language with a longer history and even more widespread acceptance. Although it is flow-chart-oriented rather than statement-oriented, the basic process concepts expressed here apply to it.

A key feature of process-orientation is that a single program is made to act as though it is several programs, independently controlled either by activity-type scans or event scheduling. Each process has several points at which it interacts with other processes. Each process can have several active phases; each active phase of a process is an event. This is different from pure event or activity approaches that allow an interaction only when all the actions associated with an event or activity have been completed, e.g., when they return to the timing mechanism.

The programming feature that makes this scheme possible is the reactivation point, which is essentially a pointer that tells a process routine where to start execution after some time-delay command has been executed. Figure 3 illustrates the concepts of interaction point and reactivation point for prototype event, activity and process routines.

In Fig. 3a, there is one reactivation point and one interaction point. An event routine always starts at the same executable statement, and, while it may have several physical RETURN statements, only one can be executed in any activation. When it is executed, it returns control to the master control program, which selects the next event (previously scheduled) to occur. All actions taken within the event routine take

	EVENT ARRIVAL	routine declaration
reactivation point →		
	SCHEDULE AN ARRIVAL AT 100.0	creation of a future interaction point
	actions to change system state	
	RETURN	interaction with other events takes place when routine returns to control program
	END	

(a) Prototype event routine

	ACTIVITY BERTHING	routine declaration
reactivation point →		
	tests to determine if act can occur	activity tests
	actions taken during berthing	executed if tests indicate activity can occur
	RETURN	interaction with other activities when routine returns to control program
	END	

(b) Prototype activity routine

	PROCESS SHOPPING	routine declaration
reactivation point →	actions to start shopping	
	WAIT 15 MINUTES	interaction point
reactivation point →	actions to shop	
	WAIT UNTIL SERVER IS FREE	interaction point
reactivation point →	actions to check out	
	DELAY 10 MINUTES	interaction point
reactivation point →	actions to return home	
	SCHEDULE SHOPPING IN 15 MINUTES	creation of future interaction point
	actions to renew shopping process	
	END	interaction point

(c) Prototype process routine

Fig. 3 -- Concepts of interaction point and reactivation point

place at the same simulated time, independently of other events. The event is not totally divorced from other events, as all events share the same system data.

In Fig. 3b, there is again only one reactivation point and one logical interaction point. If an activities test section permits them to take place, all actions occur at the same simulated time.

Figure 3c presents a sharply different picture, with many reactivation and interaction points.

Figures 3a, b, and c show that reactivation and interaction points always come in pairs. A minute's reflection will show that this has to be so. At each interaction point a reactivation point is defined, which is the place execution will start when the indicated time delay elapses or the condition being sought occurs. Within a process routine, all actions do not necessarily take place at the same simulated time, but through a series of active and passive phases.

The reader should be able to see the differences among the event, activity, and process prototypes and get a qualitative feel for how the three differ.

Each modeling scheme has distinct virtues. Each can be shown to be advantageous in some situations and disadvantageous in others. There are no rules for selecting one scheme over another in given situations, nor is it likely that any such rules will ever be stated. The universe of possible simulation models is so large and so diverse that there would undoubtedly have to be more exceptions than firm rules. Several points, however, are clear:

A language employing event scheduling gives a modeler precise control over the execution of programs.

A language employing activity scanning simplifies modeling multi-resource systems by allowing conditional statements of resource availability to be specified in one place.

A process-oriented language reduces the number of "overhead" statements a programmer has to write, since he can combine many event subprograms in one process routine. In addition, the overall flow of a system is clear, as all logic is contained in one routine rather than several.

On the other hand, there is nothing one scheme can do that another cannot. Questions of feasibility must be separated from questions of efficiency. Also, as more experience is gained with languages employing these schemes, more efficient algorithms will be developed and efficiency, per se, will become less of a problem. Eventually, modeling esthetics will become an overriding consideration.

Table 4 categorizes many of the SPLs used today according to the dynamic modeling scheme they employ.

Table 4

SPL DYNAMIC MODELING SCHEMES

Event-oriented Languages	Activity-oriented Languages[a]	Process-oriented Languages
GASP	AS [51]	GPSS
SEAL [56]	CSL	NSS [50]
SIMCOM [58]	ESP [65]	OPS [27]
SIMPAC [2]	FORSIM-IV	SIMPLE [17]
SIMSCRIPT	GSP	SIMULA
SIMTRAN [5]	MILITRAN [48]	SLANG [32]
	SILLY [57]	SOL [41]
	SIMON [31]	SPL [52]

[a]Some of these languages are not "pure," e.g., GSP and MILITRAN have both activity-scan and event-selection phases. The principal orientation is as indicated, however.

III. SIMULATION PROGRAMMING LANGUAGE FEATURES

SPECIFYING SYSTEM STRUCTURE

Every SPL must have some way of describing system structure in both its static and dynamic aspects. Section II discussed the principal features needed for this; Table 5 summarizes them.

Table 5

SYSTEM MODELING FEATURES

Statements to:
- Define classes of objects within a system
- Adjust the number of objects within a class as system conditions change
- Define properties of system objects
- Describe relationships among system objects
- Define activities, events, or processes
- Organize events or processes

Programs to:
- Select activity, event, or process subprograms for execution
- Advance simulation time

REPRESENTING STATISTICAL PHENOMENA

To model the real world, one must have a way of modeling random factors and effects. It is necessary to model uncertainty and variability with equal ease.

Uncertainty enters into models in statements such as:

> In situation X, 15 percent of the time Y will occur and 85 percent of the time Z will occur. Given that a system is in state X, some probabilistic mechanism is required to select either state Y or state Z as the next state.

Variability enters into models in statements such as:

> The time to travel from A to B has an exponential distribution with a mean of 3 hours, or the number of customers expected to arrive per hour has a Poisson distribution with a mean of 6. A probabilistic mechanism must be available for generating samples from statistical distributions.

In reproducing variability or uncertainty, a simulation model must have a way of generating random variables. A basic feature of every SPL is a random-number generator. Additional features are programs that transform random numbers into variates from various statistical distributions and perform related sampling tasks.

A process is random if predictions about its future behavior cannot be improved from a knowledge of its past behavior. A sequence of numbers is a random sequence if there is no correlation between the numbers, i.e., if there is no way to predict one number from another. Random numbers are needed to introduce uncertainty and variability into models, but because of the kinds of experiments that are performed with simulation models, truly random sequences of numbers are not adequate. One must have reproducible sequences of numbers that are, for all intents and purposes, random so far as their statistical properties are concerned.

Pseudorandom numbers, as reproducible streams of randomlike numbers are called, are generated by mathematical formulae in such a way that they appear to be random. Since they are not random, but come from deterministic series, they can only approximate the independence of truly random number sequences. Every simulation study calls for verification of random-number generators to insure that the statistical properties are adequate for the experiment being performed [20]. Every SPL must have a procedure for generating statistically acceptable sequences of pseudorandom numbers.

Pseudorandom number sequences always consist of numbers that are statistically independent and uniform [illegible] istributed between 0 and 1. Generation of a pseudorandom number [illegible] es a real number somewhere in this range.

Pseudorandom numbers can be used directly for statistical sampling tasks. They can represent probabilities in a decision sense or in a sampling sense. The model statement:

Make decision D_1 60 percent of the time,
Make decision D_2 40 percent of the time,

can be implemented in an SPL by generating a pseudorandom number and testing whether it lies between 0.0 and 0.60. If it is, decision D_1 is taken; if it is not, decision D_2 is taken. For a sufficiently large number of samples, D_1 will be selected 60 percent of the time, but the individual selections of D_1 or D_2 will be independent of previous selections.

The model statement:

Produce product P_1 20 percent of the time,
Produce product P_2 10 percent of the time,
Produce product P_3 15 percent of the time,
Produce product P_4 20 percent of the time,
Produce product P_5 35 percent of the time,

can be implemented in a similar way by sampling from a cumulative probability distribution. A random product code can be drawn from the above product mix by putting the product frequency data into a table such as

Product type	Cumulative probability
1	0.20
2	0.30
3	0.45
4	0.65
5	1.00

In this table, the difference between the successive cumulative probability values is the probability of producing a particular product; e.g., product 3 is produced 0.45 - 0.30 = 0.15 or 15 percent of the time. When a pseudorandom number is generated and matched against the table, a random product selection is made. For example, generating the number 0.42 selects product 3. Since numbers between 0.30 and 0.45 will be generated 15 percent of the time, 15 percent of the product numbers generated will be type 3.

While this type of sampling is useful for empirical frequency distributions, it is less useful for sampling from statistical distributions such as the exponential and normal. To use a table look-up procedure such as the one described above, and sample accurately in the tails of a statistical distribution, large tables must be stored. Generally, a simulation cannot afford the tables, needing the storage for model data and program. Algorithms rather than table look-up procedures are used.

Sampling algorithms are of many kinds. Some distributions are easily represented by exact mathematical formulae, some must be approximated. All sampling methods operate in the same way insofar as they transform a pseudorandom number to a number from a particular statistical distribution. References 10, 49, and 63 discuss such procedures in detail. As simulation is almost always performed using sampling, procedures that can generate samples from standard statistical distributions are mandatory in an SPL.

In conducting sampling experiments, which is what simulations really are, one is interested in control and precision as well as accuracy of representation. The topics dealt with so far have all been concerned with representation.

Control is necessary when one is using simulation to test and compare alternative rules, procedures, or qualities of equipment. When several simulation runs are made that differ only in one carefully altered aspect, it is important that all other aspects remain constant. One must be able to introduce changes only where they are desired. This is one of the reasons for requiring reproducible random-number streams. A feature that aids in this is the provision of multiple streams of pseudorandom numbers. Having more than one stream enables parts of a model to operate independently, as far as data generation is concerned, and not influence other parts. For example, when studying decision rules for assigning men to jobs, one does not want to influence the generation of jobs inadvertently. Multiple pseudorandom number streams increase a programmer's control over a model.

One also wants to be able to control the generation of random numbers if doing so can reduce the variability of simulation generated performance figures. For example, it is always desirable to make the variance of the estimate of the average length of a waiting line within a simulation model as small as possible. The reduction of sample variance is a statistical rather than a programming problem in all but one respect; a programmer should be able to control the generation of pseudorandom numbers if this is required. One known way to reduce variance is to use antithetic variates in separate simulation runs; this is discussed in [20]. As the generation of a stream of variates that are antithetic to a given stream involves no more than a simple subtraction,* this feature should be present in an SPL.

Table 6 summarizes the minimum statistical sampling features an SPL should have:

Table 6

STATISTICAL SAMPLING FEATURES

- Pseudorandom number generation
  - Multiple random-number streams
  - Antithetic variates
- Sampling from empirical table look-up distributions
- Sampling from theoretical statistical distributions

DATA COLLECTION, ANALYSIS, AND DISPLAY

The performance of a simulated system can be studied in several ways [34]. The dynamics of the system's behavior can be traced by looking at plots of relevant simulation variables as they change over time. The aggregate performance can be studied by looking at statistical analyses of simulation generated data; means, variances, minima, maxima, and histograms are usually produced for such summaries.

Ideally, an SPL should automatically produce all data collection, analysis, and display. Unfortunately, this cannot always be done, since format requirements differ among organizations, and display media vary; what is possible on a plotter may not be possible on a line printer or a typewriter. Also, efficiencies are gained if certain data are not analyzed. There is no virtue in producing frequency counts of variables that are not of direct interest to a simulation experimenter.

*If r is a generated pseudorandom number, its antithetic variate is 1 - r.

There are several topics to discuss in this general area: how data collection is specified, what data collection facilities should be provided, how display media can be used, how display fo. .s are specified, and what data analyses should be performed.

Data Collection Specification

The best one can say of a data collection specification is that it is unobtrusive. While data collection is necessary, statements that collect data are not per se part of a simulation model's logic and should not obscure the operations of a model in any way. People find that debugging is difficult enough without having to deal with errors caused by statements intended only to observe the behavior of a model.

The ultimate in unobtrusiveness is to have no specification statements at all. Being free from them clearly eliminates any difficulties they may cause when reading or debugging a simulation program code. Unfortunately, having no specification at all means that every possible piece of data must be collected in every possible way, at the risk of neglecting to collect something an analyst may want. In small models this is probably worthwhile. In large models it can lead to unacceptable increases in core storage requirements and program running times. GPSS collects certain data automatically and allows a programmer to collect other data himself; GASP does something similar.

A reasonable alternative is a linguistically natural set of data collection statements that can be applied globally to a model. Being linguistically natural, they will be easy to use and clearly differentiable from other types of programming statements. Being globally applicable, they need be written only once, rather than at each place a particular item of data to be collected appears.

Barring this, data can be collected through explicit procedural program statements. Data-collection specification statements of this sort are no different from normal variable assignment statements or subroutine calls. They are the easiest to implement in an SPL, but the most obtrusive and difficult to deal with. Most SPLs provide facilities of this kind. SIMSCRIPT II [39] has a capability for global data-collection specification.

Data Collection Facilities

One must be able to collect a variety of data, since one should be able to compute all the statistics an analyst might want about a simulation variable. This includes counts of the number of times a variable changes value, sums, sums of squares, maxima and minima of these values, histograms over specified intervals, cross-products of specified variables, time-integrated sums and sums of squares for time-dependent data, and time series displays. Simulation is a statistical tool, and statistically useful data are required to use it.

Naturally, some data are easier to collect than others. Table 7 lists the minimum data one should be able to collect.

Table 7

DATA COLLECTION FEATURES

Number of observations, maxima, and minima for all variables
Sums and sums of squares for time-independent variables
Time-weighted sums and sums of squares for time-dependent variables
Variable value histograms for time-independent variables
Time-in-state histograms for time-dependent variables
Time series plots over specified time intervals

These data should be easily collectable with specialized statements. One should be able to collect any other data without extreme difficulty. An important feature of an SPL is that it allow reasonably free access to all model data.

Data Analysis

One should not have to program the analysis of data for standard statistical calculations, such as the computations of means and variances. If global specifications are employed, names attached to statistical quantities should invoke calculations when the names are mentioned. If data collection statements are used, standard functions should operate on named data to compute the necessary quantities.

Table 8 shows the minimum analysis one should be able to perform from collected data. If the data are present, one would also like to have functions that compute correlation coefficients and spectra [21].

Table 8

DATA ANALYSIS FEATURES

Means
Variances and standard deviations
Frequency distributions

Display Media

Standard statistical information is easily printed on typewriters and line printers. Time series plots and histograms are enhanced by graphic display. As this type of information derives most of its impact from visual observation, there is little reason it should not be presented this way. Advanced SPLs should have routines for charting results, either by simulating a plotter on a line printer or by displaying results directly on a plotter [13], [23], [62].

Today, with a growing number of large-scale computing systems making use of cathode ray tube displays (CRTs), these devices are being used more and more for displaying simulation output [54]. Two situations lend themselves to CRT application.

In the first situation, the CRT is used only to produce attractively formatted graphs and reports. The device is not viewed on-line; pictures are made and used in lieu of printed reports. There is no doubt that programmers can use enhanced graphical capabilities if given the opportunity. Generally, no changes need be made to a SPL to let them do so, other than providing access to general system software routines. To be specific, a programmer should be able to call upon library plotting routines from a SIMSCRIPT or GPSS program.

The second situation is the more glamorous, with output produced on-line as a program is executed. Given a language and an operating system that lets a programmer interrupt a running program, alter system parameters and variables, and then continue simulating where he left off, an entirely new type of simulation debugging and experimentation is possible. This type of interactive, adaptive dialogue between model and programmer makes on-line, evolutionary model design possible, changes the economics of sequential, optimum-seeking experimentation, and adds a valuable dimension to program debugging. Several researchers, at The RAND Corporation and elsewhere, are currently working in this area [17], [30].

Specification of Display Formats

There are probably as many types of output statements as there are people who write programming languages. Each type, being a little different, emphasizes one or more aspects of output control at the expense of others. Styles range from no specification at all (GPSS), through format-free statements (SIMSCRIPT II) and formatted statements (CSL), to special report forms (SIMSCRIPT). There are times when each style has its merits, and a fully equipped SPL will have a variety of output display statements.

Four types of display statements that exist in present-day SPLs are:

(1) Automatic output in a standard format (GPSS, GASP):

Is a time-saver for the programmer and a boon in reasonably small models where all data can be displayed at a reasonable cost.

Does not force a beginning simulation programmer to deal explicitly with output.

Is only as good as the exhaustiveness of its contents.

Is often unsatisfactory for formal reports, forcing subsequent typing and graph preparation.

(2) Format-free output (SIMSCRIPT II):

Enables a programmer to control the display of information without regard for formats.

Is adequate only if it covers all the data structures in a language.

Is most useful for debugging, error message reporting, and printing during program checkout.

(3) Formatted output (CSL):

Requires the most programmer knowledge, but provides the maximum control of information display.

Is traditionally the most difficult part of many programming languages, insofar as the greatest number of errors are made by novice programmers in format statements.

(4) Report Generators (SIMSCRIPT, GSP):

Are the easiest way of producing specially designed reports.

Must have a complete complement of control facilities to cover all report situations.

Can be a nuisance to use in very simple situations.

Usually generate an extremely large amount of object code. Are efficient from a programming standpoint, but not from a core-consumption point of view.

Since the production of reports is the primary task of all programs, whether they are run for checkout, for display of computed results, or for preparation of elaborate management reports and charts, a good SPL should contain statements adapted to all display situations. Going back to the discussion of data collection, a programmer should not have to spend a great deal of his time writing output statements. He should be able to concentrate on model construction and programming and not have to dwell at length on conventional output tasks. He should be able to spend time on sophisticated output statements, however, to produce displays that are unusual or that deal with exotic display devices.

MONITORING AND DEBUGGING

Two essential requirements of all SPLs can be served by the same set of programming facilities. SPLs should be able to assist in:

(1) Program debugging; and in

(2) Monitoring system dynamics.

Debugging can be difficult in high-level programming languages, as there is generally a great deal of difference between source and object codes. Errors can be detected during compilation and execution that are only distantly related to source-language commands. Moreover, when an SPL is translated into an intermediate POL, as was originally done in SIMSCRIPT and CSL, execution error messages are often related to the intermediate language and not the programmer's source statements. These messages, while meaningful to an expert, can mislead a novice SPL programmer.

Debugging is also difficult because the flow of control in a simulation is stochastically determined. Moreover, it can be difficult to obtain a record of the flow of control, since an SPL-designed "timing routine" or other form of control program is the originating point for all event calls. In some languages, it is impossible to do so. Without program flow information, and information about the system state at various times, some simulation program errors can be found only by luck.

The debugging features an SPL should provide are listed in Table 9.

Table 9

DEBUGGING FEATURES

Report compile and execute-time errors by source statement related messages;

Display complete program flow status when an execute-time error occurs. This means displaying the entry points and relevant parameters for all function and subroutine calls in effect at the time of the error;

Provide access to control information between event executions. This allows event-tracing during all or selected parts of a program, and use of control information in program diagnostic routines.

These same facilities are needed for monitoring system dynamics. As one use of simulation is the study of system behavior, one must be able to view sequences of events and their relevant data to observe system reactions to different inputs and different system states. Event-tracing is an important tool for this kind of study.

In an event-oriented SPL, debugging and monitoring features will undoubtedly be implemented differently from the same or similar features in activity- or process-oriented SPLs. This is not important. The basic issue is whether some basic facility exists for assisting in program debugging and for doing program monitoring.

INITIALIZATION

Because simulation is the movement of a system model through simulated time by changing its state descriptions, it is important that an SPL provide a convenient mechanism for specifying initial system states. In simulations dedicated to studying start-up or transient conditions, a convenient mechanism for doing this is mandatory; in simulations that only analyze steady-state system performance, it is still necessary to start off at some feasible system configuration.

Some SPLs start simulation in an "empty and idle state" as their normal condition and require special efforts to establish other conditions. They rely either on standard input statements, formatted or unformatted, to read in data under program control or on preliminary programs that set the system in a predetermined state.

An alternative to these procedures is a special form that reduces the initialization task to filling out a form rather than writing a program. While adequate in a large number of situations, this alternative suffers from being inflexible. As with the preparation of simulation reports, the correct answer lies in a mixture of initialization alternatives.

Another aspect of initialization is the ability to save the state of a system during a simulation run and reinitialize the system to this state at a later time. This facility is crucial in the simulation of extremely large systems, and in conducting sequential experiments. One should be able to save all the information about a program, including relevant data on the status of external storage devices and peripheral equipment, and restore it on command at a later date.

OTHER FEATURES

There are a number of non-operational features that must be taken into consideration when designing or selecting an SPL. A manager or analyst is interested in program readability; communication of the structure, assumptions and operations of a model is important if the model is to be used correctly. A manager is also interested in execution efficiency; simulations can require large numbers of experimental runs, and the cost per run must be low enough to make a project economical. On the other hand, a manager must balance the costs of producing a program against program execution costs. Complex modeling languages may compile and execute less efficiently than simpler languages, but they make problems solvable in a shorter period of time. If total problem solving time is important rather than computer time costs, the evaluation criteria change.

SPL documentation is important to applications and system programmers. An applications programmer needs a good instruction manual to learn a language and to use as a reference guide. As an SPL becomes more complex the need for good documentation increases. Systems programmers need documentation to be able to maintain an SPL. This documentation must allow them to install the language in the computing system; with today's complex, hand-tailored systems this is becoming more difficult. It must also provide enough information for them to make modifications in the SPL itself as translator errors are discovered. It is less important that users be able to modify an SPL, either to change the form of statements or to add new ones, but this can be an important consideration in certain instances. Some languages in fact are designed to do this easily [16], [38], [50].

IV. SOME EXAMPLES OF SPLS

This section illustrates four SPLs: SIMSCRIPT II, an event-oriented language; SIMULA, a process-oriented language; CSL, an activity-oriented language; and GPSS, a transaction-oriented language.* With the exception of the SIMSCRIPT II example, which appears for the first time in this Memorandum, the illustrations are taken from published descriptions of the respective languages. The examples differ in detail and specificity, but are nevertheless representative of the concepts the languages employ. As they have been taken from other sources with only a surface editing, they also differ greatly in style and format.

Because the SIMSCRIPT II example was written especially for this Memorandum, it is the most detailed and illustrates the greatest number of features. Consequently, the reader may tend to judge SIMSCRIPT II a superior language. He should not make such a judgment solely on the basis of these examples. Ideally, they should all be comparable and not bias the sought-after end, which is the explication of their different approaches to providing the SPL features discussed in Sec. III. We can only hope that our inability to procure "equally representative" examples will not detract from our purpose.

The concepts the languages employ have all been described in previous sections and readers should be able to follow the examples without a thorough understanding of each. The format of the following subsections is: description of a model, simulation program for the model, discussion of the program.

SIMSCRIPT II: AN EVENT-ORIENTED LANGUAGE

The model used in this example is the "executive-secretary system" described in [34]. The program conforms as closely as possible to the description given in [34] and the flowcharts of its events.

---

*Although GPSS is process-oriented, in the sense that its models take a synoptic view of systems, its basic orientation is with the flow of transactions rather than the occurrence of processes.

The Model

We assume that executives in an office system have two types of tasks: they process incoming communications (invoices, requests for bids, price queries) and handle interoffice correspondence. The tasks are not independent of one another; the former are produced by mechanisms external to the office system, the latter arise during daily operations. As they result in similar actions we can treat both the same way, through an event that "discovers" a task. Other events assign tasks to secretaries, schedule coffee breaks and departures for lunch, and handle the review of completed secretarial tasks. Table 10 lists the objects that "live" in the office system -- which we shall call entities from here on -- and their attributes.

Table 10

SYSTEM ENTITIES AND THEIR ATTRIBUTES

Executive	Secretary	Task
Position: Manager Senior Junior State: Busy Available On break	Skill in typing: words/minute errors/100 words Skill in dictation: words/minute errors/100 words Skill in office work: general rating 1-100 State: Busy Available On break	Type: Invoice Price quotation Bid Telephone Dictation Typing Characteristics: Time Secretarial requirements Probability of requiring a follow-up task

Given the static structure defined in Table 10, the nature of the task-discovery event, and some logic not yet described, we can construct a flowchart model of the actions that take place when a request enters the system. This model is illustrated in Fig. 4. Numbers to the left of each flowchart block refer to comments in the body of the text that describe the operations that take place within the block. The SIMSCRIPT II program for the model follows the

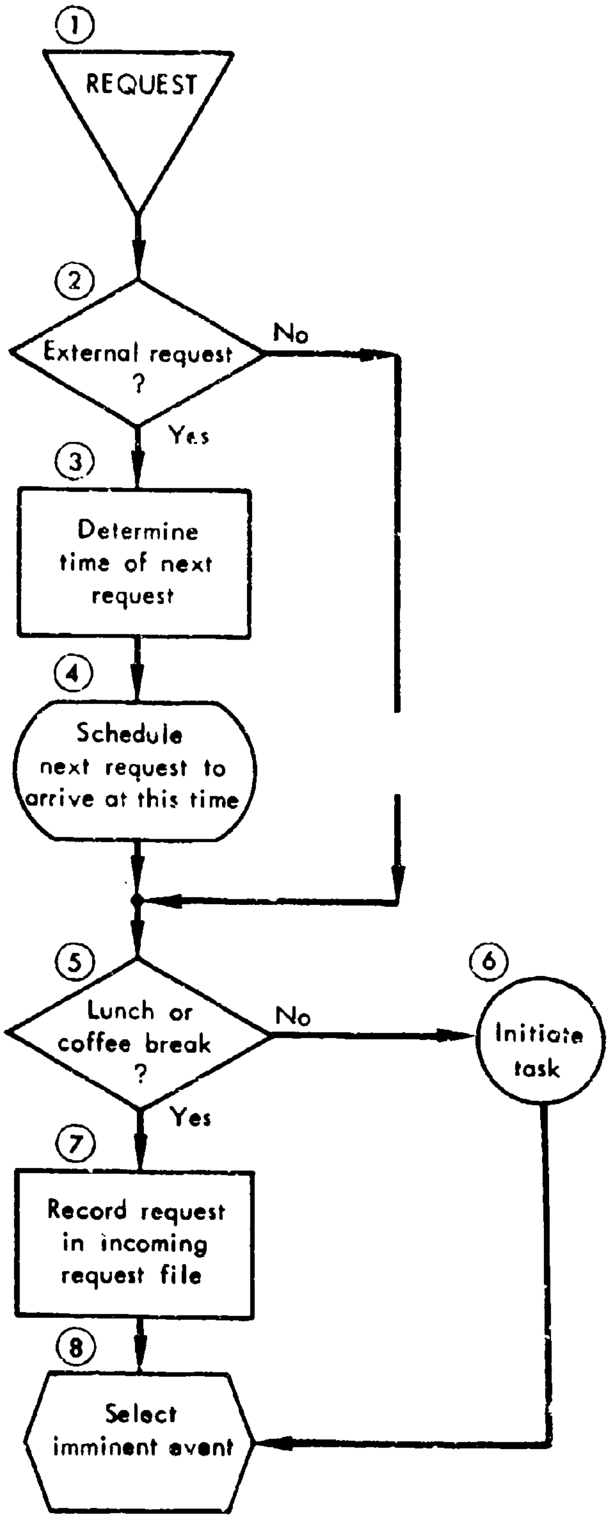

1
REQUEST
2
External request ?
No
Yes
3
Determine time of next request
4
Schedule next request to arrive at this time
5
Lunch or coffee break ?
No
Yes
6
Initiate task
7
Record request in incoming request file
8
Select imminent event

Fig. 4 -- Event number 1: request to perform a task

flowcharts and their description. The flowcharts and their respective programs differ somewhat as the flowcharts are simplified for the sake of clarity.

Block 1 is the entry point to the flowchart. It contains a name that will be used in subsequent flowcharts to refer to the "task request" event. The directed arrow leading from it is a symbol commonly used to indicate a path and direction of flow.

Block 2 is a decision block that splits the logical flow, depending on the kind of request that has just occurred. To understand how this block operates we must understand the concept of an event occurrence.

An event occurs when its "time arrives," the time having been previously recorded by an internal scheduling block or observed on an input data card. The precise mechanism that accomplishes these tasks need not be stated here. It suffices if the reader understands that there is some mechanism operating in the background of the simulation program, observing data cards and previously scheduled events, ordering them by their event times, and "popping them up" when their time arrives. This in fact is the function of the event selection block (Block 8). The reader will notice that every event terminates with an event selection block. It is in event selection blocks that time discriminations are made, events selected, and the simulation clock advanced.

When a request event is popped up, the simulation program has access to information associated with it, e.g., how it was caused. The model is able to look at this information and take action on it.

If the request is for an internally generated task, the flowchart leads directly to Block 5, where a question is asked to see if office workers are available to process the request. If the request is for an externally generated task, the program pauses in Blocks 3 and 4 to read information about the next arrival from an external data source, and schedule its arrival at some future time. When it does so it records a memo of a request arrival and its time on a calendar of events scheduled to occur. This calendar is part of the selection mechanism employed in sequencing events and advancing simulation time. These operations are performed by the SIMSCRIPT II system and do not have to be programmed explicitly.

By the time the program arrives at Block 5, it is through with scheduling future events and is concerned with processing the request that has just arrived. Since real offices do not work continuously, but pause for lunch and coffee breaks during the day, the model asks if such a period is in progress. If it is, the request cannot be processed immediately but must be filed for later handling. If the request can be processed, Block 6 transfers control to a routine that does so. The routine will return to the request event when it finishes processing the task.

Block 7 records a request that cannot be handled in a backlog file; it might be an in-basket in real life. The file entry is made so that when the office workers return to their desks they see the tasks that accumulated while they were gone.

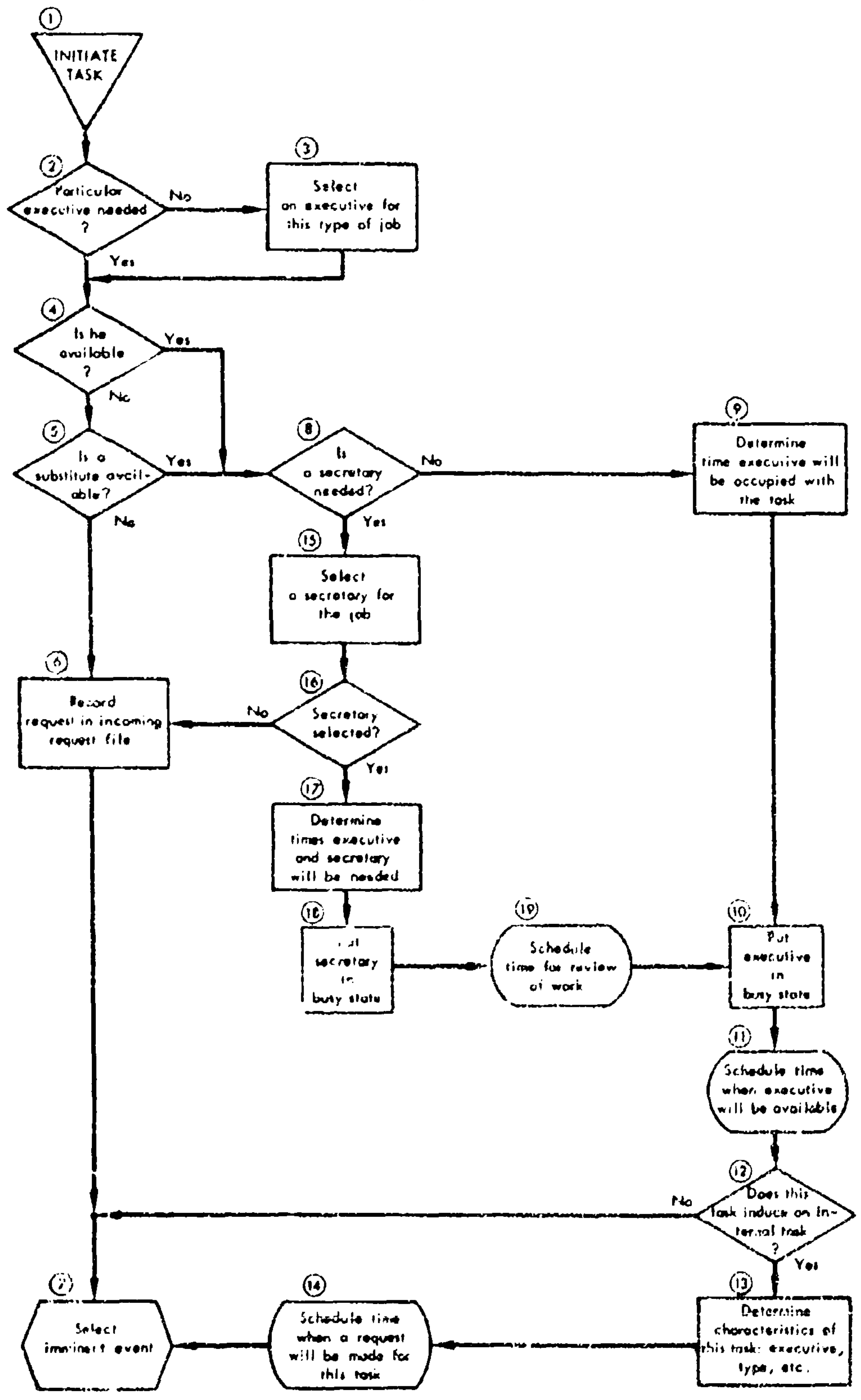
Block 8 directs the simulation program to select an event from the time-ordered file of scheduled events. It might be another request or the completion of a previous task. When the next event is selected, it may or may not indicate a simulation time advance. If it does not, we think of it and the event just completed as occurring simultaneously; although they are processed in series on the computer, there is no time advance and they are considered as happening at the same time.

Initiating a Task

Once the system has accepted a request, a match must be made between it and the resources needed to fill it. A routine is written to do this; its logic is shown in Fig. 5. First a search is made for an executive. If one is found who is free and can handle the request, a secretary is procured if needed.

Block 1, as always, is an entry block giving the symbolic name of the routine.

Block 2 starts the match between a task and its resources by asking if the request just entered calls for a particular executive, e.g., there has been a telephone call for a certain person or a request for a price quotation from a specialist in a certain area. If no particular executive is called for, Block 1 passes flow to Block 3, where an executive is selected. If a certain person is requested, flow proceeds to Block 4, where a test is made to see if this person is available.



1 INITIATE TASK
2 Particular executive needed ?
No
Yes
3 Select an executive for this type of job
4 Is he available ?
Yes
No
5 Is a substitute available?
Yes
No
8 Is a secretary needed?
No
Yes
9 Determine time executive will be occupied with the task
15 Select a secretary for the job
6 Record request in incoming request file
16 Secretary selected?
No
Yes
17 Determine times executive and secretary will be needed
18 Put secretary in busy state
19 Schedule time for review of work
10 Put executive in busy state
11 Schedule time when executive will be available
12 Does this task induce an internal task ?
No
Yes
7 Select imminent event
14 Schedule time when a request will be made for this task
13 Determine characteristics of this task: executive, type, etc.

Fig. 5 -- Ini[illegible]ion of a task routine

Block 3 is typical of a functional block whose description is short but whose programming content might be large. A procedure to select an executive can be brief, e.g., managers can do everything, senior executives can do everything except give price quotations, junior executives can only answer the telephone; or it can be long and elaborate, e.g., an executive is selected whose personal qualifications as listed in his personnel file match the requirements of the task according to a complex and computationally intricate formula. Many of a simulation model's key assumptions are built into blocks such as this.

When an executive is selected, Block 3 transfers to Block 4, the block to which control is passed if a particular executive is called for.

Block 4 asks if the executive requested in Block 2 or selected in Block 3 is available. It does so by examining the executive's state (status code); if the code is "available," the executive is free to handle the request, if it is "busy" or "on break," he is not. Once again, as in Block [illegible], the flow logic is split depending on the answer to this question.

If the selected executive is available, flow passes to Block 8, where processing of the task continues. Before we consider these actions we should discuss what happens if the executive is not available.

Block 5 asks if a substitute is available for a busy executive, implying that a substitution can be made and that a procedure exists for finding one. This situation is a little like that of Block 3, where an executive is selected for a particular type of task. Block 5 could be expanded to a series of blocks describing a procedure for selecting a substitute, testing for his availability, selecting another substitute if necessary, and so on until all possible candidates are tried and accepted or rejected. In our simplified model we do not do this. We only indicate that if a substitute cannot be found, control passes to Block 6, which files the unprocessed task.

Block 6 of this event is identical to Block 7 of the request event; it files information about the request for later processing. This block appears in the simulation model whenever a request cannot be processed and must be "remembered."

In Block 7 control is passed via an event selection block back to the "timekeeping" mechanism of the simulation program. Since the current request cannot be processed, the model must look at its calendar of scheduled events to determine what to do next.

Returning to the case where an executive is available to process a request, we ask next in Block 8 if a secretary is also needed. She will be if the request is for dictation or for some task where instructions must be given; she will not be if the task is simply answering a telephone call. This question can be answered in a number of ways in an operating computer program; as with most questions of this type, we leave the description of decision-making at the macro level, namely, that a decision has to be made.

Block 9 starts the flow path for the case where a request can be honored by an executive alone; it determines the amount of time he will spend on the task.

Block 10 puts the executive in a "busy" state so that he cannot be called on to do another task while he is working on this one. He will remain in this state until the "executive available" event occurs; this is scheduled in Block 11 to happen after the lapse of the previously determined amount of time.

Before proceeding with the simulation, the model must ask if processing this task, e.g., answering a phone call, induces another task, e.g., writing a memo. This is done in Block 12. If a task is not induced, flow passes to Block 7, where the model is instructed to select another event and proceed with the simulation. If a task is induced, Block 13 determines its characteristics and passes them on to Block 14, where the induced task is scheduled to be requested. Flow then proceeds to Block 7.

If, back in Block 8, we found that a secretary was needed to work along with the executive, control would have passed to Block 15, where a secretary must be selected before a task starts. This logic can pair a particular secretary with an executive, pool all secretaries so that they are available to all executives, or employ some immediate scheme. As was done in selecting an executive, when a secretary is chosen, her status code must be tested to see if she is available.

Block 16 performs this test. Like Block 5, it can be considered a macro block in which alternatives and availabilities are tested until a decision is reached. If a secretary is not available, a request cannot be processed and must be filed along with other unprocessed requests.

Once a secretary is found, Blocks 17, 18, and 19 determine the time the executive and secretary will spend on the job, put the secretary in the "busy" state, and schedule the time when her work will be reviewed. It is not necessary that the executive and the secretary work together on the task for the same period of time; separate events are provided to schedule their release from the task at different times. The release times can be the same, however, if the task is a cooperative effort.

Block 19 transfers control to Block 10 after completing its function, picking up at a part of the flowchart that we have already seen. The reader should be able to see why and how this is done.

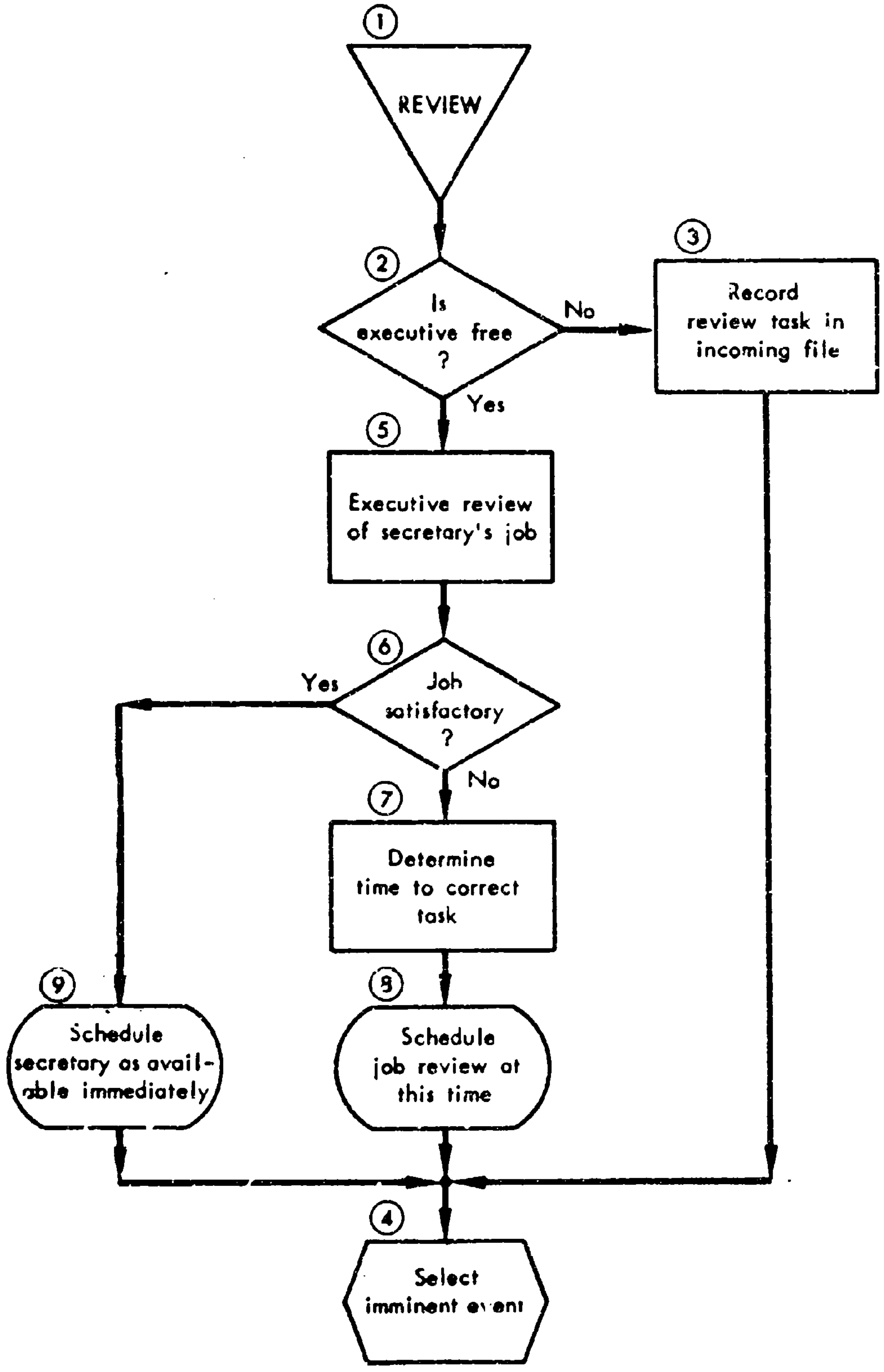
Review of a Secretarial Task

One of the office rules is that every task a secretary performs must be reviewed. When a secretary finishes a task she brings it to the attention of the executive who initiated it. If he is not available, she waits.* If he is available, he reviews the work and either accepts it or notes corrections that must be made before another review. The logic of the review event is shown in Fig. 6.

Block 1, as usual, names the event. Block 2 asks a question about executive availability and transfers to Block 3 or 5, depending on the answer.

Block 3 records the review task in the task backlog file if the executive is busy. The task is filed along with incoming requests that were filed for reasons we saw in previous flowcharts. Block 4 calls on the simulation timing mechanism to select the next scheduled

*This may not be good office practice, but is a feature of our example.



1
REVIEW
2
Is executive free ?
No
3
Record review task in incoming file
Yes
5
Executive review of secretary's job
6
Job satisfactory ?
Yes
No
7
Determine time to correct task
9
Schedule secretary as available immediately
8
Schedule job review at this time
4
Select imminent event

Fig. 6 -- Event Number 2: Review of a secretarial task

event. The secretary is not assigned an "available" state, but remains "busy," waiting for the executive to become free and review her work.

Block 5 is another macro block, hiding what might be an enormous amount of logic behind the label "executive review of secretary's work."

Block 6 branches on the previously computed review decision. If the task has been done satisfactorily, the secretary is scheduled to become available immediately (Block 9).

An unsatisfactory task has its correction time computed in Block 7 and review rescheduled in Block 8.

Concerning this event, it is important to note its hidden basic assumptions: a review task takes no time and a secretary stays with a job until it gets reviewed. These assumptions can be easily changed to allow secretaries to do other work while waiting for reviews, cause executives to spend time making large-scale corrections, and so on.

Executive Available at the End of a Task

This event marks the completion of an executive activity. It returns an executive to an "available" state and determines his next action: another task, a break for coffee or lunch, or an idle (discretionary time) period. The event logic is shown in Fig. 7.

Block 1 names the event. Block 2 puts the executive in an available state and asks questions about the next executive actions. These questions are asked in a specific order and imply certain things. From the logic of the event, we see that a lunch or coffee break cannot start until a current job is completed, but will be taken when it is due regardless of task backlogs. This is important as it assumes a priority sequence imposed by the order in which questions are asked and not by explicit priority statements.

Block 3, which decides if a break is due, also contains hidden logic. When one considers the connections between events and the way in which the model operates, he sees that if an executive is idle (in the "available" state) and a break time occurs, there is no mechanism

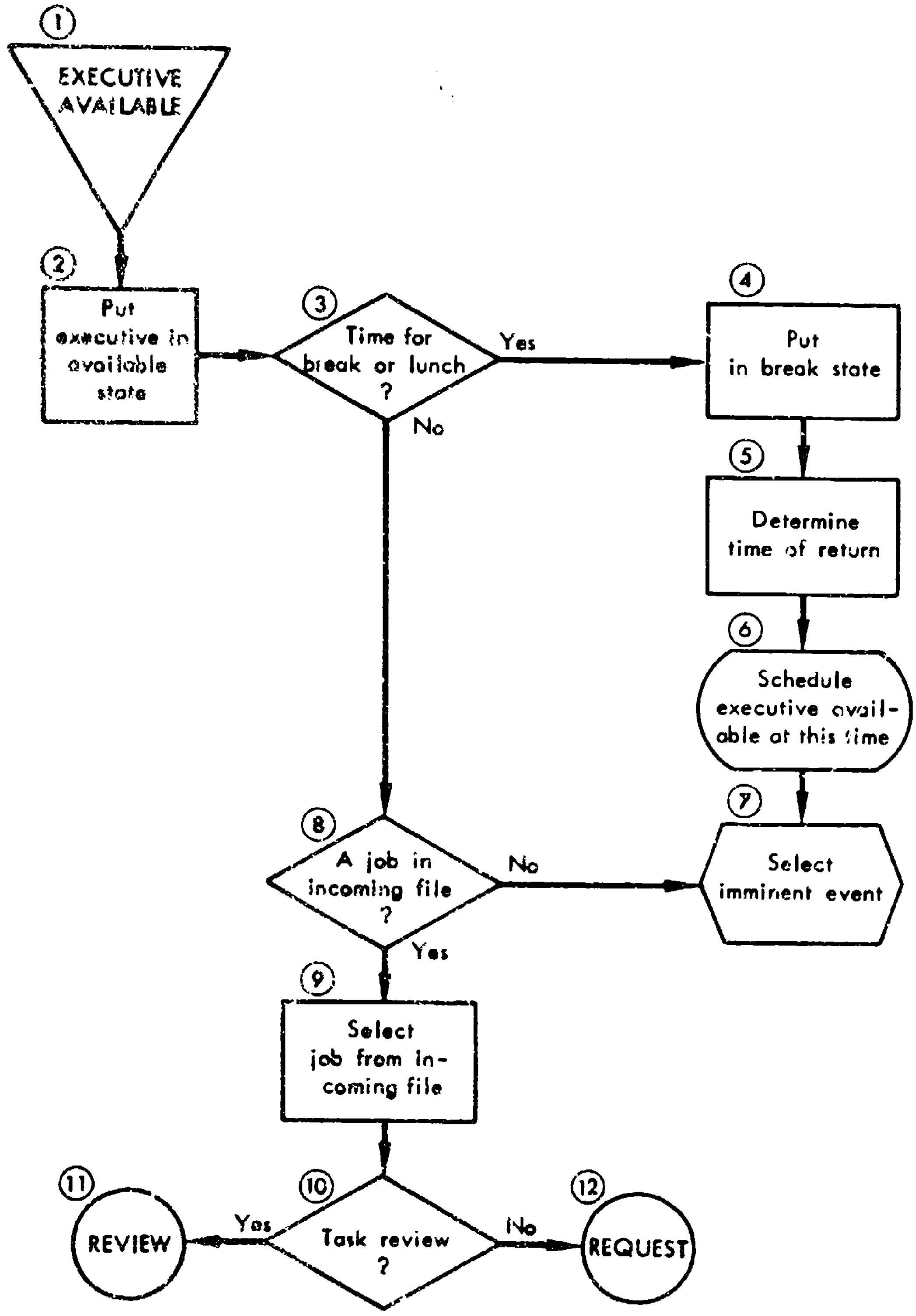


Fig. 7 -- Event Number 3: Availability of an executive at the end of a task

that alerts him of this. By the way the model is constructed, breaks can be taken only after the completion of jobs. This will have little practical effect if: (a) the work rate is high in the office so that there are no long periods of idle time possible, or (b) the logic of Block 3 looks ahead and starts a break early if one is almost due. This small difficulty has been put in the model to acquaint the reader with problems that can occur when one sets out to build a model from scratch.

If a break is due or in progress, Block 4 places the executive in the "break" state, Block 5 determines its duration, Block 6 schedules the executive's return to an availability condition (by executing this same event some time in the future after the simulation clock has advanced past the break point), and Block 7 returns control to the event selection mechanism.

If a break is not due, the model must decide whether the executive should be left in the available state or assigned to a waiting task. It does this by looking, in Block 8, at the file in which we have been putting requests that could not be processed. If the file is empty, the executive is left alone and control is passed to Block 7 to select the next event.

If the file is not empty, the model must select a job. If there is only one job in the file, there is no problem. If there is more than one, there is a conflict situation that must be resolved. Conflict is usually resolved by priority rules that assign values to different types of jobs; a job is selected that has the highest (or perhaps lowest) value. In cases with ties, multiple ranking criteria are used. Possible criteria that might be used in this model are: time a job arrives in the system, skill level required to process a job, etc. The issue of selection rules is complex, and a model that merely says "select a job" hides a great deal of work that must be done to develop an operating model. For example, few organizations have well-articulated and formalized priority rules, and a modeler may have his hands full merely trying to find out the "rules of the game."

Once a task has been selected, however, it is relatively simple to route the executive to the proper flowchart to process it. This is shown in Blocks 10, 11, and 12.

Secretary Available at the End of a Task

This event is similar to event 3 in both its intent and its form. When a secretary is released from a task, she becomes available and is either sent on a break, put on a backlogged job, or left idle, depending on current conditions. Blocks 1 through 10 in the flowchart of Fig. 8 correspond to similar blocks in Fig. 7 and need not be commented upon here.

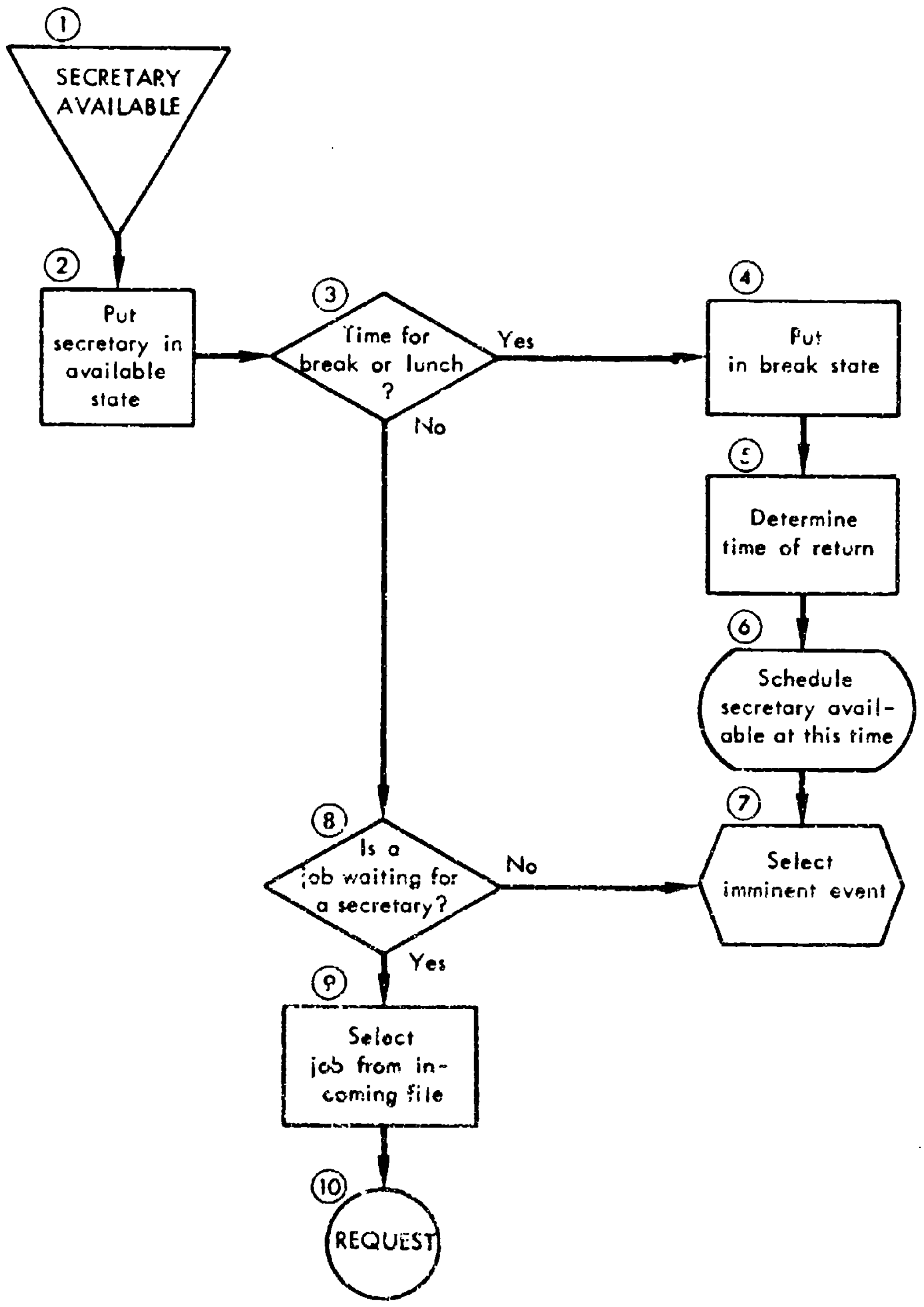

1
SECRETARY AVAILABLE
2
Put secretary in available state
3
Time for break or lunch ?
Yes
No
4
Put in break state
5
Determine time of return
6
Schedule secretary available at this time
7
Select imminent event
8
Is a job waiting for a secretary?
No
Yes
9
Select job from incoming file
10
REQUEST

Fig. 9 -- Event number 4: availability of a secretary at the end of a task

<u>The Program</u>

```
 PREAMBLE
 NORMALLY MODE IS INTEGER
 ''DECLARATION OF STATIC SYSTEM STRUCTURE
 THE SYSTEM OWNS A REQUEST.FILE
 PERMANENT ENTITIES....
    EVERY EXECUTIVE HAS A POSITION AND A STATE
    EVERY SECRETARY HAS A STATUS
    EVERY TASK.TYPE HAS A TASK.TIME, A NEED, AND
       AN INDUCE.PROBABILITY
    EVERY SECRETARY, TASK.TYPE HAS A SKILL.FACTOR
       DEFINE TASK.TIME, INDUCE.PROBABILITY AND
          SKILL.FACTOR AS REAL VARIABLES
 TEMPORARY ENTITIES....
    EVERY TASK HAS A WHO, A WHAT AND A DELAY.TYPE
       AND MAY BELONG TO THE REQUEST.FILE
 ''DECLARATION OF DYNAMIC SYSTEM STRUCTURE
 EVENT NOTICES....
    EVERY REQUEST HAS AN EXEC AND A CLASS
    EVERY REVIEW HAS AN EX AND A SEC
    EVERY EXECUTIVE.AVAILABLE HAS AN EX1
    EVERY SECRETARY.AVAILABLE HAS A SEC1
 EXTERNAL EVENTS ARE REQUEST AND END.OF.SIMULATION
 BREAK REVIEW TIES BY HIGH EX THEN BY LOW SEC
 PRIORITY ORDER ''OF EVENTS'' IS REQUEST, REVIEW, SECRETARY.AVAILABLE,
    EXECUTIVE.AVAILABLE AND END.OF.SIMULATION
 ''OTHER DECLARATIONS
 DEFINE OFFICE.STATUS AS AN INTEGER FUNCTION
 DEFINE INDUCE.TYPE, SECRETARY.REQUIRED.CLASS AND
    SUM.REQUESTS AS INTEGER VARIABLES
 DEFINE SATISFACTORY AS A REAL VARIABLE
 DEFINE IDLE TO MEAN 0
 DEFINE WORKING TO MEAN 1
 DEFINE BREAK TO MEAN 2
 ''DATA COLLECTION AND ANALYSIS DECLARATIONS
 ACCUMULATE AVG.BACKLOG AS THE MEAN AND
    STD.BCKLOG AS THE STD.DEV OF N.REQUEST.FILE,
 ACCUMULATE STATE.EST(0 TO 2 BY 1)AS THE HISTOGRAM
    OF STATE
 ACCUMULATE STATUS.EST(0 TO 2 BY 1)AS THE HISTOGRAM
    OF STATUS
 END
```

```
           MAIN ''THIS ROUTINE CONTROLS THE SIMULATION EXPERIMENT
'INITIALIZE' CALL INITIALIZATION ''TO ESTABLISH THE INITIAL SYSTEM STATE
           START SIMULATION ''BY SELECTING THE FIRST EVENT
           ''WHEN SIMULATION RUN IS ENDED CONTROL PASSES HERE
           IF DATA IS ENDED, STOP
           OTHERWISE...''GET SET FOR ANOTHER RUN
              UNTIL REQUEST.FILE IS EMPTY, DO
                 REMOVE THE FIRST TASK FROM THE REQUEST.FILE
                 DESTROY THE TASK
              LOOP
           GO INITIALIZE ''FOR THE NEXT EXPERIMENT
           END
```

```
           ROUTINE FOR INITIALIZATION
           READ N.EXECUTIVE    CREATE EACH EXECUTIVE
           READ N.SECRETARY    CREATE EACH SECRETARY
           READ N.TASK.TYPE    CREATE EACH TASK.TYPE
           FOR EACH EXECUTIVE, DO READ POSITION(EXECUTIVE) AND
              STATE(EXECUTIVE) RESET TOTALS OF STATE LOOP
           FOR EACH SECRETARY, DO
              READ STATUS(SECRETARY) RESET TOTALS OF STATUS
              ALSO FOR EACH TASK.TYPE, READ SKILL.FACTOR(SECRETARY, TASK.TYPE),
                 TASK.TIME(TASK.TYPE), NEED(TASK.TYPE),
                 INDUCE.PROBABILITY(TASK.TYPE)
           LOOP
           READ INDUCE.TYPE, SECRETARY.REQUIRED.CLASS AND SATISFACTORY
           RESET TOTALS OF N.REQUEST.FILE
           END
```

```
           EVENT REQUEST GIVEN EXEC AND CLASS SAVING THE EVENT NOTICE
           ADD 1 TO SUM.REQUESTS ''COUNT NUMBER OF TASK REQUESTS
           IF REQUEST IS EXTERNAL, READ EXEC AND CLASS ''FROM A DATA CARD
           REGARDLESS...''PROCESS THE REQUEST
           IF OFFICE.STATUS=WORKING,
              NOW INITIATE.TASK GIVING EXEC AND CLASS    GO AHEAD
           OTHERWISE...''FILE THE REQUEST UNTIL THE BREAK IS OVER
              CREATE A TASK     ''TO ACT AS A MEMO
              LET WHO=EXEC      ''RECORD WHO THE REQUEST WAS FOR
              LET WHAT=CLASS    ''RECORD THE TYPE OF TASK
              LET DELAY.TYPE=0 ''RECORD THAT THE MEMO REPRESENTS A REQUEST
               ''RECEIVED DURING A BREAK PERIOD
              FILE THE TASK IN THE REQUEST.FILE
'AHEAD'    DESTROY THE REQUEST
           RETURN ''TO THE TIMING ROUTINE
           END
           ROUTINE TO INITIATE.TASK GIVEN EXECUTIVE AND CLASS
           DEFINE EXEC.TIME AND SEC.TIME AS REAL VARIABLES
           IF EXECUTIVE,  ''NO EXECUTIVE HAS BEEN SPECIFIED, SELECT ONE
              FOR EACH EXECUTIVE WITH STATE=IDLE AND POSITION GE NEED(CLASS),
                 FIND THE FIRST CASE
              IF FOUND, GO TO 'SEC.TEST'
              ELSE...''NO EXECUTIVE AVAILABLE FOR THIS TASK
'NO.EXEC'        LET D=1  ''INDICATING THE TASK IS WAITING FOR AN EXECUTIVE
'NO.WORKER'      CREATE A TASK    LET WHO=0    LET WHAT=CLASS
                    LET DELAY.TYPE=D
                 FILE THE TASK IN THE REQUEST.FILE
                 RETURN  ''TO THE TIMING ROUTINE
           ELSE...''AN EXECUTIVE HAS BEEN REQUESTED

           IF STATE¬=IDLE,
              ''REQUESTED EXECUTIVE IS BUSY, LOOK FOR SUBSTITUTE
              CALL SUBSTITUTION GIVING EXECUTIVE YIELDING EXECUTIVE
           THEN IF EXECUTIVE=0, ''NO SUBSTITUTE CAN BE FOUND
              GO TO NO.EXEC
           ELSE ''AN EXECUTIVE IS AVAILABLE, IS A SECRETARY REQUIRED?
'SEC.TEST' IF CLASS >= SECRETARY.REQUIRED.CLASS,
              ''A SECRETARY IS REQUIRED FOR THIS TASK
              PERFORM SECRETARY.SELECTION YIELDING SECRETARY
              IF SECRETARY=0, ''NO SECRETARY IS AVAILABLE FOR TASK
                 LET D=2 ''INDICATING THE TASK IS WAITING FOR A SECRETARY
                 GO TO NO.WORKER
              ELSE...''DETERMINE TIME EXECUTIVE AND SECRETARY WORK
              LET EXEC.TIME=EXPONENTIAL.F(TASK.TIME(CLASS),1)
              LET SEC.TIME=EXEC.TIME + EXPONENTIAL.F(TASK.TIME(CLASS),1)*
                                         SKILL.FACTOR(SECRETARY,CLASS)
              LET STATUS=WORKING ''SET THE SECRETARY IN THE WORKING STATE
              SCHEDULE A REVIEW(EXECUTIVE, SECRETARY) IN SEC.TIME MINUTES
           REGARDLESS...
           IF EXEC.TIME=0 ''EXECUTIVE IS WORKING ALONE AND MUST COMPUTE
            ''HIS TIME
              LET EXEC.TIME=2*EXPONENTIAL.F(TASK.TIME(CLASS),1)
           REGARDLESS...
           LET STATE=WORKING ''SET THE EXECUTIVE IN THE WORKING STATE
           SCHEDULE AN EXECUTIVE.AVAILABLE(EXECUTIVE) IN EXEC.TIME MINUTES
           IF CLASS > INDUCE.TYPE, ''CHECK FOR AN INDUCED TASK
              CREATE A REQUEST CALLED INDUCED
              LET EXEC(INDUCED)=EXECUTIVE
              LET CLASS(INDUCED)=CLASS-1
              SCHEDULE THE REQUEST CALLED INDUCED IN UNIFORM.F(0.0,1.0,1) HOURS
           REGARDLESS
           RETURN ''TO THE TIMING ROUTINE
           END
```

```
ROUTINE SUBSTITUTION GIVEN EXEC YIELDING EXEC1
''FIND THE FIRST IDLE EXECUTIVE WITH AT LEAST THE SAME RANK
FOR EACH EXECUTIVE WITH STATE=IDLE AND POSITION ≥
   POSITION(EXEC), FIND EXEC1=THE FIRST EXECUTIVE
IF NONE, LET EXEC1=0
REGARDLESS    RETURN ''TO THE CALLING PROGRAM
END
```

```
ROUTINE FOR SECRETARY.SELECTION YIELDING SECRETARY
''FIND THE FIRST IDLE SECRETARY
FOR EACH SECRETARY WITH STATUS=IDLE, FIND THE FIRST CASE
IF NONE, LET SECRETARY=0
REGARDLESS    RETURN ''TO THE CALLING PROGRAM
END
```

```
            EVENT REVIEW GIVEN EXECUTIVE AND SECRETARY
            IF STATE¬=IDLE, ''EXECUTIVE BUSY, CANNOT REVIEW JOB
               CREATE A TASK
               LET WHO=EXECUTIVE    LET WHAT=SECRETARY
               LET DELAY.TYPE=3 ''INDICATING A DELAYED REVIEW
               FILE THE TASK IN THE REQUEST.FILE
               DESTROY THE REVIEW
               GO RETURN
            ELSE...''EXECUTIVE REVIEWS SECRETARY'S WORK
            IF RANDOM.F(2) LE SATISFACTORY,
               ''TASK HAS BEEN PERFORMED SATISFACTORILY
               SCHEDULE A SECRETARY.AVAILABLE(SECRETARY) NOW
               GO RETURN
            ELSE...''TASK MUST BE CORRECTED
               RESCHEDULE THIS REVIEW IN 15 MINUTES
'RETURN'    RETURN ''TO THE TIMING ROUTINE
            END
```

```
          EVENT EXECUTIVE.AVAILABLE GIVEN EXECUTIVE
          LET STATE=IDLE ''PUT EXECUTIVE IN THE IDLE STATE
          IF OFFICE.STATUS ¬=WORKING,
             ''A BREAK PERIOD IS IN PROGRESS
             LET STATE=BREAK ''PUT THE EXECUTIVE IN THE BREAK STATE
             RESCHEDULE THIS EXECUTIVE.AVAILABLE AT TRUNC.F(TIME.V)+1
             GO RETURN
          OTHERWISE...''EXECUTIVE IS FREE TO WORK ON BACKLOGGED TASKS
          IF REQUEST.FILE IS EMPTY, GO RETURN
          ELSE...''FIND TASKS NEEDING EXECUTIVE ATTENTION
          FOR EACH TASK IN THE REQUEST.FILE WITH DELAY.TYPE ¬=2
             FIND THE FIRST CASE ''NOT WAITING FOR A SECRETARY
          IF NONE, GO RETURN ''NO BACKLOGGED EXECUTIVE JOBS
          ELSE...''EXAMPLE TASK
             REMOVE THE TASK FROM THE REQUEST.FILE
          IF DELAY.TYPE=0 OR DELAY.TYPE=1, ''WAIT IS TO START A NEW TASK
             CREATE A REQUEST    SUBTRACT 1 FROM SUM.REQUESTS
             SCHEDULE THE REQUEST(WHO, WHAT)NOW
             GO AHEAD
          ELSE ''TASK IS A SECRETARY REVIEW, THE VARIABLE "WHAT"
               ''IS USED FOR THE SECRETARY IDENTIFICATION
          SCHEDULE A REVIEW(WHO, WHAT)NEXT
'AHEAD'   DESTROY THIS TASK
'RETURN'  RETURN ''TO THE TIMING ROUTINE
          END
```

```
          EVENT SECRETARY.AVAILABLE GIVEN SECRETARY
          LET STATUS=IDLE ''PUT SECRETARY IN THE IDLE STATE
          IF OFFICE.STATUS ¬ =WORKING,
             ''A BREAK PERIOD IS IN PROGRESS
             LET STATUS=BREAK ''PUT THE SECRETARY IN THE BREAK STATE
             SCHEDULE THIS SECRETARY.AVAILABLE AT TRUNC.F(TIME.V)+1
             GO RETURN
          ELSE...''SECRETARY IS FREE TO WORK ON BACKLOGGED TASKS
          IF THE REQUEST.FILE IS EMPTY, GO RETURN
          ELSE...''FIND TASKS NEEDING SECRETARIAL ATTENTION
          FOR EACH TASK IN THE REQUEST.FILE WITH DELAY.TYPE=2,
             FIND THE FIRST CASE
          IF NONE, GO RETURN ''NO TASKS WAITING FOR A SECRETARY
          ELSE...
          REMOVE THE TASK FROM THE REQUEST.FILE
          CREATE A REQUEST    SUBTRACT 1 FROM SUM.REQUESTS
          SCHEDULE THE REQUEST(WHO, WHAT)NOW
          DESTROY THIS TASK
'RETURN'  RETURN ''TO THE TIMING ROUTINE
          END
          EVENT END.OF.SIMULATION
          NOW REPORT
          FOR I=1 TO EVENTS.V, ''EMPTY THE EVENTS LIST
          UNTIL EV.S(I) IS EMPTY, DO
             REMOVE THE FIRST J FROM EV.S(I)
             GO TO REQ OR REV OR SEC OR EXEC PER I
             'REQ' DESTROY THE REQUEST CALLED J
                   GO LOOP
             'REV' DESTROY THE REVIEW CALLED J
                   GO LOOP
             'SEC' DESTROY THE SECRETARY.AVAILABLE CALLED J
                   GO LOOP
             'EXEC' DESTROY THE EXECUTIVE.AVAILABLE CALLED J
'LOOP'    LOOP
          RETURN ''TO THE TIMING ROUTINE
          END



          ROUTINE FOR OFFICE.STATUS
          DEFINE T AS A REAL VARIABLE
          LET T=MOD.F(TIME.V,24)
          IF 12 = HOUR.F(TIME.V) OR 10.75 < T < 11 OR
             15.75 < T < 16, RETURN WITH 0 ''INDICATING BREAK IN PROGRESS
          ELSE RETURN WITH 1 ''INDICATING OFFICE NOW WORKING
          END



          ROUTINE REPORT
          START NEW PAGE
          PRINT 2 LINES WITH AVG.BACKLOG AND STD.BACKLOG THUS
             AVERAGE BACKLOG IS **.** TASKS
             STD.DEV         IS  *.**
          SKIP 3 OUTPUT LINES
          BEGIN REPORT
          BEGIN HEADING
          PRINT 2 LINES THUS
             ANALYSIS OF EXECUTIVE STATUS
                IDLE    WORKING    BREAK
          END ''HEADING
          FOR EACH EXECUTIVE, PRINT 1 LINE WITH STATE.EST(EXECUTIVE,1)/
             TIME.V, STATE.EST(EXECUTIVE,2)/TIME.V, STATE.EST(EXECUTIVE,3)/
                TIME.V AS FOLLOWS
                *.**      *.**       *.**
          END ''REPORT
          SKIP 3 OUTPUT LINES
          BEGIN REPORT
          BEGIN HEADING
          PRINT 2 LINES THUS
             ANALYSIS OF SECRETARY STATUS
                IDLE    WORKING    BREAK
          END ''HEADING
```

```
FOR EACH SECRETARY, PRINT 1 LINE WITH STATUS.EST(SECRETARY,1)/
   TIME.V, STATUS.EST(SECRETARY,2)/TIME.V, STATUS.EST(SECRETARY,3)/
      TIME.V AS FOLLOWS
      *.**    *.**    *.**
END ''REPORT
SKIP 5 OUTPUT LINES
PRINT 1 LINE WITH SUM.REQUESTS AND TIME.V LIKE THIS
   ***REQUESTS WERE PROCESSED IN ****.* SIMULATED DAYS
RETURN ''TO CALLING PROGRAM
END
```

Description of the Program

Rather than describe the SIMSCRIPT II program in detail, we discuss only those statements that highlight SPL features mentioned in previous sections. The purpose of the examples is to show how various languages implement simulation programming concepts, not to describe the languages themselves. Those who wish to understand the examples and the languages more fully can do so by studying their respective programming manuals.

The preamble to the subprograms that make up the SIMSCRIPT II model declares the static system structure of the model, using the entity-attribute-set organization framework; declares the events that compose the dynamic structure; defines special properties of the two structures, such as the mode of attributes, the ranking of sets, and the priority order of events; and specifies data-collection and analysis tasks. The preamble is a set of global declarations that describe the system being simulated to the SIMSCRIPT II compiler. In the case of the data-collection and analysis statements and some debugging statements not illustrated in the example, the preamble also specifies tasks that the compiler is to perform.

The main routine provides overall simulation experiment control. It calls on a programmer-written routine to initialize the static system state and provide events for the timing routine that will set the model in motion. The START SIMULATION statement removes the first scheduled event (the initialized event with the earliest event time) from the file of scheduled events and starts the simulation by transferring program control to it.

Eventually, either by running out of data or by programmer action, all events are processed, no new ones are created, and control passes from the timing routine (represented by the START SIMULATION statement) to the statement that follows it. If there are no more data, the sequence of experiments is terminated. If there are more data, the system is initialized for another run.

The two routines MAIN and INITIALIZATION illustrate the primary features SIMSCRIPT II provides for controlling simulation experiments.

In the next routine, an event named REQUEST, the features of interest are the statements CREATE, FILE, and DESTROY. The CREATE statement generates a new entity of the class TASK whenever it is executed; this statement is SIMSCRIPT II's way of dynamically allocating storage to system entities as they are needed. The DESTROY statement takes a named entity, REQUEST in this example, and returns it to a pool of free data storage, providing space for the subsequent creation of additional entities. The FILE statement takes a named entity and puts it in a set along with other entities. In this example an entity named TASK is put in a set named REQUEST.FILE.

In the routine INITIATE.TASK, the features to note are IF and FOR statements that perform logical tests and searches, the statistical function EXPONENTIAL.F, and the event-scheduling statements. The IF and FOR statements are SIMSCRIPT II's way of dealing with the common programming problem of determining the state of objects, or of the system itself, and selecting among objects according to stated criteria. The statistical function indicates the way sampling is done to represent statistically varying phenomena. The last argument in the EXPONENTIAL.F function-call selects one of ten built-in number streams; if the argument is negative the antithetic variate of the generated pseudorandom number is used. The SCHEDULE statements are the basic mechanism for specifying events that are to occur in the future. When a SCHEDULE statement is executed, an entity, called an event notice, of a specified type is put in a time-ordered file that is ordered by the schedule time; when the simulation clock advances to this time, the event is executed and is said to "occur."

The event routines EXECUTIVE.AVAILABLE and SECRETARY.AVAILABLE contain REMOVE statements. These statements retrieve entities from sets according to criteria that are either implied or specified in the preamble. One of the functions of the preamble is to specify such things as the relationship that entities in sets have to one another.

The REPORT routine illustrates SIMSCRIPT II's facilities for generating reports. No output is collected, analyzed, or printed automatically by SIMSCRIPT II. Rather, the data-collection and analysis statements of the preamble and the report specification features pictured are used to tailor reports to simulation experiment requirements.

Naturally, this brief explanation has not made the program clear in all its details -- that was not its intent. Rather, its purpose is to show how SIMSCRIPT II provides the simulation-oriented features discussed in Sec. III. All three of the following examples follow this same pattern.

SIMULA: A PROCESS-ORIENTED LANGUAGE

This example has been taken from Chapter 13 of [13].* Its model is similar to those used in many SPL descriptions, and aside from terminology, structurally very similar to the SIMSCRIPT II model just presented. The program is quite different.

The Model

A job consists of machine groups, each containing a given number of identical machines in parallel. The system is described from a machine point of view, i.e., the products flowing through the system are represented by processes that are passive data records. The machines operate on the products by remote accessing.

The products consist of orders, each for a given number of product units of the same type. There is a fixed number of product types. For each type there is a unique routing and given processing times.

For each machine group (number mg) there is a set avail[mg] of idle machines and a set que[mg], which is a product queue common to the machines in this group. The products are processed one batch at a time. A batch consists of a given number of units, which must belong to the same order. The batch size depends on the product type and the machine group.

A product queue is regarded as a queue of orders. The queue discipline is essentially first-in-first-out, the position of an order in the queue being defined by the arrival of the first unit of that order. However, if there is less than an acceptable batch of units of a given order waiting in the queue, i.e., if the batch size is too small as yet, the next order is tried. The last units of an order are accepted

*Courtesy of the Norwegian Computing Center.

as a batch, even if the number of units is less than the ordinary minimum batch size. If a machine finds no acceptable batch in the product queue, it waits until more units arrive.

Although the individual pieces of product are "units," a unit is not treated as an individual item in the present model. For a given <u>order</u> and a given <u>step</u>, i.e., machine group, in its schedule, we define an <u>opart</u> (order part) record to represent the group of units currently involved in that step. The units are either in processing or waiting to be processed at the corresponding machine group.

An order is represented by a collection of opart records. The sum of units in each opart is equal to the number of units in the order. Each opart is a member of a product queue. If a machine group occurs more than once in the schedule of a product type, there may be more than one opart of the same order in the product queue of that machine group.

Among the attributes of an opart record are the following integers: the order number, <u>ono</u>, the product <u>type</u>, the <u>step</u>, the number of units waiting, <u>nw</u>, and the number of units in processing, <u>np</u>. The flow of units in the system is effected by counting up and down the attributes nw and np of opart records.

An opart record is generated at the time when the first batch of units of an order arrive at a machine group. It is entered at the end of the corresponding product queue. The opart will remain a member of this queue until the last unit has entered processing. It will drop out of the system when the last unit has finished processing. A Boolean attribute <u>last</u> is needed to specify whether a given opart contains the last units of the order involved in this step.

At a given time the units of an order may be distributed or several machine groups. There will be an opart record* for each of them. An opart process* will reference the one at the next step, i.e., machine group, through an <u>element</u> attribute "<u>successor</u>." An order is thus represented by a simple chain of opart records. The one at the

*The terms "record" and "process" both refer to the data structure associated with a particular group of units. See Lines 5-7, p. 66.

head has no successor, the one at the tail has its attribute "last" equal to true. The chain "moves" through the system by growing new heads and dropping off tails.

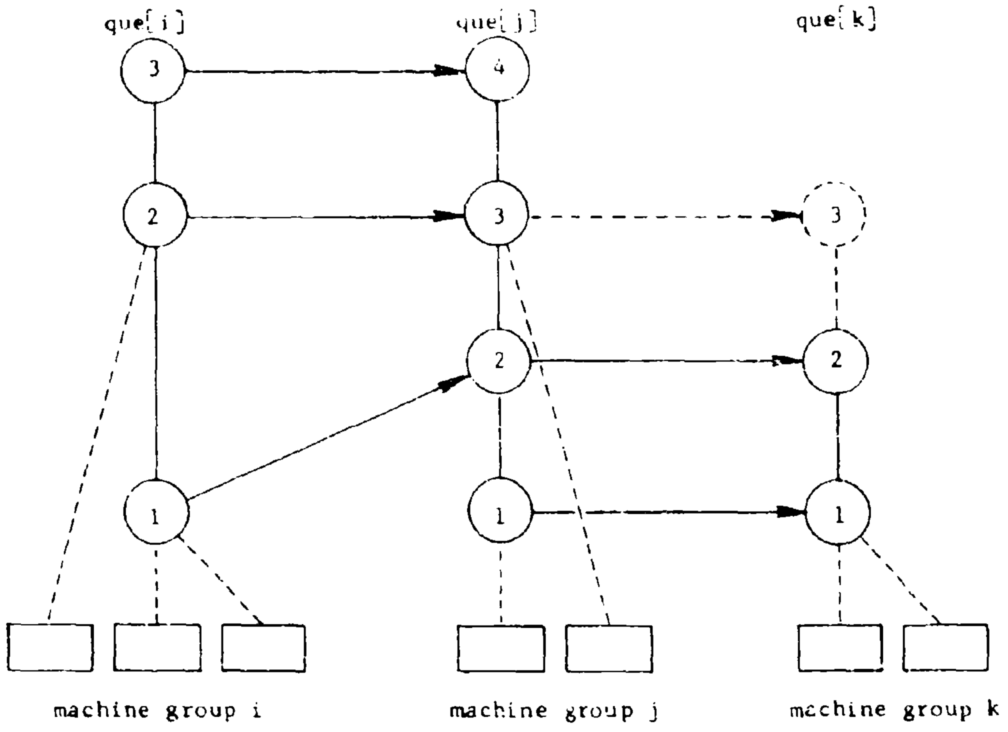


Fig. 9 -- Flow of products through the shop

Figure 9 shows three consecutive steps in the schedule of products of a given type. A product queue consists of oparts (circles) connected by vertical lines. Oparts belonging to the same order are connected by horizontal lines. Machines are represented by squares. A dotted line between an opart and a machine indicates a batch of units in processing. When the batch of the third opart in que[j] is finished, a new opart receiving this batch will be generated and included in que[k].

The Program

The following program fragment is part of the head of a SIMULA block describing the above system. A machine activity is given. For clarity, only statements essential for the behavior of the model are shown. The program is not complete. Underlined words are SIMULA keywords.

```
set array que, avail [1:nmg]; integer U;
integer procedure nextm (type, step); integer type, step;....;
real procedure ptime (type, step); integer type, step;....;
integer procedure bsize (type, mg); integer type, mg;....;
activity opart (ono, type, step, nw, np, last, successor);
  integer ono, type, step, nw, np;
  Boolean last; element successor;
activity machine (mg); integer mg;
  begin integer batch, next; Boolean B; element X;
serve: X:=head (que[mg]);
  for X:=suc (X) while exist (X) do
  inspect X when opart do
  begin batch :=bsize (type, mg);
    if nw < batch then begin
      if last then batch :=nw else go to no end;
    nw :=nw - batch; np  :=np + batch;
    if last∧nw = 0 then remove (X);
    activate first (avail[mg]);
    hold (batch x ptime (type, step)xuniform (0.9, 1.1, U));
    np := np - batch; B := last∧nw + np = 0;
    next := nextm (type, step);
    inspect successor when opart do
       begin nw := nw + batch; last := B end
    otherwise begin successor :=
       new opart (ono, type, step + 1, batch, 0, B, none);
       include (successor, que [next]) end;
    activate first (avail [next]);
    go to serve;
no: end;
    wait (avail [mg]); remove (current); go to serve end;
```

Description of the Program

Line 1	The sets contain oparts and idle machines, respectively. The variable U defines a pseudorandom number stream (line 19).
Lines 2-4	The functions "nextm" and "ptime" specify the next machine group and the current processing time for a given product type and step in the schedule. "bsize" determines the batch size, given the product type and machine group number. The three functions are left unspecified, i.e., their programs are not shown.
Lines 5-7	The meanings of the attributes of opart processes have been explained in the model description. The activity body[*] is a dummy statement: an opart process is a data record with no associated actions.
Line 8	The machine activity extends to and includes line 30. The parameter mg is the machine group number. Machines belonging to the same group are completely similar.
Line 9	"batch" is the size of the current batch of units, "next" is the number of the next machine group for the units currently being processed, the meaning of "B" is explained below (line 20), and "X" is used for scanning.
Line 10	Prepare for scanning the appropriate product queue. Select the first opart in que[mg].
Line 11	Scan. The controlled statement is itself a connection statement[**] (lines 12-29).

[*] In the SIMULA nomenclature, a process is a dynamic structure of an activity, i.e., an activity is a process prototype.

[**] "Connection" is a means of accessing local variables from outside the block in which they are defined. In this instance, attributes of the oparts stored in que[mg] are being referenced.

Line 12	There is only one connection branch (lines 12-29). Since a product queue contains only opart records, connection must become effective. The attributes of the connected opart are accessible inside the connection block.
Line 13	Compute the standard batch size.
Lines 14, 15	A smaller batch accepted only if the opart is at the tail end of the chain. In this case "nw" is nonzero (cf. line 17), and the units are the last ones of the order. Otherwise the next opart is tried by branching to the end of the set inspection loop.
Line 16	"batch" units are transferred from the waiting state to the in-processing state by reducing nw and increasing np.
Line 17	The opart is removed from the product queue when processing has started on the last units of the order.
Line 18	The current machine has found an acceptable batch of units, and has updated the product queue. There may be enough units left for another bat^h; therefore the next available machine in this group (mg) is activated. If there is no idle machine, the set avail[mg] is empty and the statement has no effect. See also lines 27 and 30.
Line 19	The expected processing time is proportional to the number of units in the batch. The actual processing time is uniformly distributed in the interval $\pm$ 10% around the expected value. The sequence of pseudo-random drawings is determined by the initial value of the variable U.
Line 20	Processing is finished; np is reduced. The Boolean variable B gets the value <u>true</u> if and only if the last

units of an order have now been processed. In that case the connected opart should drop off the chain at this system time (see comments to line 28). It follows that B is always the correct (next) value of the attribute "last" of the succeeding opart (lines 23, 25).

Line 21	Compute the number of the machine group to receive the current batch of units.
Line 22	The element attribute "successor" is inspected. The connection statement, lines 22-26, has two branches.
Line 23	This is a connection block, executed if "successor" refers to an opart. The latter is a member of the product queue of the next machine group. It receives the processed batch of units, which are entered in the waiting state. The attribute "last" is updated. Notice that the attributes referenced in this inner connection block are those belonging to the successor to the opart connected outside (X).
Lines 24, 25	If the connected opart (X) is at the head of the chain, the value of "successor" is assumed equal to none, and otherwise branch is taken. A new opart is generated, and a reference to it is stored in "successor." The new opart has the same "ono" and "type" as the old one, and its "step" is one greater. It has "batch" units in the waiting state and none in processing. Its attribute "last" is equal to "B". Since the new opart has become the head of the chain, its "successor" should be equal to none. Notice that the initial value of "last" may well be true, e.g., if the order contains a single unit.
Line 26	The new opart is included at the end of the product queue of the next machine group.

<u>Line 27</u> The current machine has now transferred a batch of units to the product queue of next machine group. Therefore the first available machine (if any) of that group is activated. If that machine finds an acceptable batch, it will activate the next machine in the same group (line 18). This takes care of the case in which the batch transferred is larger than the standard batch size of the next machine group for this type of product.

<u>Line 28</u> The machine immediately returns to the beginning of its operation rule to look for another acceptable batch, starting at the front end of the product queue. At this point, if B is <u>true</u>, the connected opart is empty of units and will not be referenced any more. We can regard it as having dropped off the chain. It is easy to demonstrate, however, that the opart will physically leave the system, i.e., that its reference count is reduced to zero. The possible stored references are:
(1) The variable X and the connection pointer "opart" of this machine or another one of the same group. The <u>go to</u> statement leads out of the connection block, which deletes the connection pointer. X is given another value in line 10. Any other machine referencing this opart would have to be suspended in line 19, which is impossible since np is zero (cf. the second statement of line 16).
(2) Set membership in que[mg]. The opart must have been removed from the queue (by this machine or another one) since "last" is <u>true</u> and nw is now zero (line 17).
(3) The attribute "successor" of the opart preceding this one in the chain. The first opart of this order to enter the systems has no predecessor. Provided that this first one drops out when it is empty, our conclusion follows by induction (see below).

Line 29 The end of the connection block and of the statement controlled by the for clause in line 11.

Line 30 If, after having searched the entire product queue, the machine has found no acceptable batch, it includes itself in the appropriate "avail" set and goes passive. Its local sequence control remains within the wait statement as long as the machine is in the passive state. When the machine is eventually activated (by another machine: line 27 or 18), it removes itself from the "avail" set and returns to scan the product queue. The "avail" sets are operated in the first-in-first-out fashion.

The mechanism for feeding orders into the system is not shown above. This is typically done by the Main Program or by one or more "arrival" processes, which generate a pattern of orders, either specified in detail by input data, or by random drawing according to given relative average frequencies of product types and order sizes.

An arrival pattern defined completely "at random" is likely to cause severely fluctuating product queues, if the load on the system is near the maximum. The following is a simple way of rearranging the input pattern so as to achieve a more uniform load. The algorithm is particularly effective if there are different "bottle-necks" for the different types of products.

```
31.  activity arrival (type, mgl, pt);
32.    integer type, mgl; real pt;
33.  begin integer units;
34.  loop: select (units, type); id := id + 1;
35.          include (new opart (id, type, 1, units, 0, true,
                                                    none), que[mgl]);
36.          activate first (avail [mgl]);
37.          hold (ptxunits); go to loop end;
38.  procedure select (n, type); value type; integer n, type; ...;
39.  integer id;
```

Line 31 — There will be one "arrival" process for each product type. "mgi" is the number of the first machine group in the schedule of this type of product. "pt" is a stipulated "average processing time" per unit, chosen so as to obtain a wanted average throughput of units of this type (see line 37).

Line 34 — The procedure "select" should choose the size, "units," of the next order of the given type, e.g., by random drawing or by searching a given arrival pattern for the next order of this type. "id" is a nonlocal integer variable used for numbering the orders consecutively.

Line 35 — An order is entered by generating an opart record that contains all the units of the order. The units are initially in the waiting state. The order is filed into the appropriate product queue. The set membership is the only reference to the opart stored by the arrival process. Consequently, this opart will leave the system when it becomes empty of units, as assumed earlier (line 28).*

Line 36 — A machine in the appropriate group is notified of the arrival of an order.

Line 37 — The next order of the same type is scheduled to arrive after a waiting time proportional to the size of this order, which ensures a uniform load of units (of each type).

The "output" of units from the system can conveniently be arranged by routing all products to a dummy machine group at the end of the schedule. It contains one or more "terminal machines" (not shown here),

*In SIMULA, process records that are no longer needed, i.e., are not referenced by any other process, are automatically returned to available storage. This contrasts with the DESTROY statement used by SIMSCRIPT for the same task.

which may perform observational functions such as recording the completion of orders.

The dynamic setup of the system is a separate task, since initially the Main Program is the only process present. The Main Program should generate (and activate) all processes that are "permanent" parts of the system, such as machines, arrival processes, and observational processes. The system can be started empty of products. However, a "steady" state can be reached in a shorter time if orders (opart records) are generated and distributed over the product queues in suitable quantities.

Experimental results are obtained by observing and reporting the behavior of the system. Three different classes of outputs can be distinguished:

(1) On-line reporting. Quantities describing the current state of the system can be printed out, e.g., with regular system time intervals: lengths of product queues in terms of units waiting, the total number of units in the system, the number of idle machines in each group, etc. A more detailed on-line reporting may be required for program debugging.

(2) Accumulated machine statistics. By observing the system over an extended period of system time, averages, extrema, histograms, etc., can be formed. Quantities observed can be queue lengths, idle times, throughputs, and so on. The accumulation of data could be performed by the machine processes themselves.

Example. To accumulate a frequency histogram of the idle periods of different lengths for individual machines, insert the following statements on either side of the "wait" statement of line 30:

"tidle := time" and "histo(T, H, time - tidle, 1)," where "tidle" is a local real variable, and T and H are arrays. T[i] are real numbers that partition observed idle periods (time - tidle) into classes according to their lengths, and

H[i] are integers equal to the number of occurrences in each class. The system procedure "histo" will increase H[i] by one (the last parameter), where i is the smallest integer such that T[i] is greater than or equal to the idle period, "time - tidle." T and H together thus define a frequency histogram, where T[i] - T[i - 1] is the width of the i'th column, and H[i] is the column length.

(3) <u>Accumulated order statistics</u>. During the lifetime of an opart record, the "history" of an order at a given machine group can be accumulated and recorded in attributes of the opart. The following are examples of data that can be found.

The arrival of the first unit of the order at this machine group is equal to the time at which the opart is generated. The departure time of the last unit is equal to the time at which the variable B gets the value <u>true</u> (line 20 of a machine connecting the opart).

The sum of waiting times for every unit of the order in this queue is equal to the integral with respect to system time of the quantity nw (which is a step function of time). The integral can be computed by the system procedure "accum." The statements "nw := nw $\pm$ batch" (lines 16 and 23) are replaced by "accum (anw, tnw, nw, $\pm$ batch)," where the <u>real</u> variables anw and tnw are additional attributes of the opart process, with initial values zero and "time," respectively. The procedure will update nw and accumulate the integral in anw. It is equivalent to the statements: anw := anw + nw x (time - tnw); tnw := time; nw = nw $\pm$ batch.

It is worth noticing that arrival times, waiting times, etc., cannot in general be found for individual units, unless the units are treated as individuals in the program. Neither can the maximum waiting time for units in an order. The average waiting time, however, is equal to the above time integral divided by the number of units in the order.

The complete history of an order in the shop is the collection of data recorded in the different oparts of the order. These data can be written out on an external storage medium at the end of the lifetime of each opart. That is, an output record could be written out before line 28, whenever B is true, containing items such as the order number, ono, the sum of waiting times, anw, the current system time, etc. When the simulation has been completed, the data records can be read back in, sorted according to order numbers, and processed to obtain information concerning the complete order, such as the total transit time, total waiting time, etc.

The same information can be obtained by retaining the complete opart chain in the system until the order is out of the shop; however, this requires more memory space. The chain can be retained by making the arrival process include the initial opart in an auxiliary set, or by having a pointer from the opart currently at the head of the chain back to the initial one. The opart chain can be processed by the terminal machine. (The order is completely through the shop at the time when the attribute "last" of the opart in the terminal product queue gets the value true.) In the former case the terminal machine should also remove the appropriate opart from the auxiliary set, in order to get rid of the opart chain.

CSL: AN ACTIVITY-ORIENTED LANGUAGE

This example has been taken from [11].* While unlike the two previous models, it is indicative of the kinds of models industrial firms construct to solve practical operating problems.

The Model

This example is a simulation of the operation of a simple port, which consists of an outer deep-water harbor and a series of berths. Each berth can hold one large ship, which can berth only at full tide, or three small ships, which can also move at half-tide. The tide runs in a 12-hour sequence, out for seven hours, half-tide for an hour.

A distribution of unloading times for large ships is available as data, and unloading times for small ships are normally distributed. Interarrival times are negative exponentially distributed.

*Courtesy of the IBM United Kingdom Data Centre.

The program is to record the waiting times of large and small ships and the times for which the berths are empty. The purpose of the simulation might be to study the operation as a basis for experiments to find a more efficient way of scheduling the working of the port, or to determine the effect of providing extra berths. The scheduling used in this model is a simple first-in first-out scheme.

The Program

```
                  PORT SIMULATION      EXAMPLE PROGRAM

      CONTROL
      CLASS TIME SHIP.100 BERTH.4
C           DEFINE CLASSES OF 100 SHIPS AND 4 BERTHS
      SET OCEAN HARBOUR LARGE SMALL FREE PART FULL
      SET SHIPIN(BERTH)
C           DEFINE THE SETS REQUIRED, INCLUDING AN ARRAY OF AS
C           MANY SETS AS THERE ARE BERTHS, SHIPIN(X) WILL HOLD
C           A LIST OF THE NAMES OF SHIPS IN BERTH X
      NAME S B
      INTEGER TIDE TLARGE TSMALL
      TIME CHANGE ARRIVE FINISH
C           DEFINE TWO NAME VARIABLES, AN INTEGER VARIABLE TO
C           SHOW THE STATE OF THE TIDE, AND ADDITIONAL TIME
C           CELLS. ALSO TWO INTEGERS TO HOLD TOTAL ARRIVALS
C           OF LARGE AND SMALL SHIPS RESPECTIVELY.
      HIST LARGEQ 25,2,5  SMALLQ 25,2,5  IDLE 25,2,5
      HIST UNLOD 20,3,5
C           DEFINE THE HISTOGRAMS REQUIRED. LARGEQ HAS 25
C           CELLS WITH RANGE 0-4 (MIDPOINT 2), 5-9,10-14 ETC.
C           UNLOD WILL CONTAIN THE UNLOADING TIME DISTRIBUTION
C           FOR LARGE SHIPS.
      INITL
      ACTIVITIES
      TIDES ARRVL BTHL BTHS DBTH ENDING
C           SPECIFY THE LIST OF SECTORS (ACTIVITIES)
      END


      SECTOR INITL
      T.FINISH=24000
      T.CHANGE=7
      T.ARRIVE=0
      TIDE=0
C           THIS SECTOR IS ENTERED ONLY ONCE AND SETS UP THE
C           INITIAL STATE OF THE MODEL. T.FINISH REFERS TO THE
C           TIME AT WHICH SIMULATION IS TO FINISH, T.CHANGE TO
C           THE TIME AT WHICH THE TIDE NEXT CHANGES AND TIDE
C           SHOWS THE STATE OF THE TIDE AS FOLLOWS -
C           0    TIDE OUT    1 HALF IN    2 TIDE FULL    3 HALF IN
C           T.ARRIVAL SHOWS THE TIME BEFORE THE NEXT ARRIVAL OF
C           A SHIP AT THE PORT.
      FOR X = 1,SHIP
        SHIP.X INTO OCEAN
      FOR X = 1,BERTH
        BERTH.X INTO FREE
        T.BERTH.X=0
```

```
C         INITIALLY ALL SHIPS ARE IN OCEAN
C         AND ALL BERTHS FREE
      READ (5,10) UNLOD
C         READ IN THE DISTRIBUTION GIVEN AS DATA.
10    FORMAT (I4)
      END


      SECTOR TIDES
C         THIS SECTOR IS CONCERNED WITH TIDE CHANGES
      T.CHANGE EQ 0
C         WHICH CAN ONLY OCCUR WHEN THEY ARE DUE
      TIDE+1
      GOTO (10,20,10,30) TIDE
C         CHANGE TIDE MARKER AND RESET TIME CELL FOR NEXT
C         CHANGE
10    T.CHANGE=1
      GOTO 60
20    T.CHANGE=3
      GOTO 60
30    T.CHANGE=7
      TIDE=0
60    DUMMY
C         AND RETURN TO CONTROL SEGMENT
      END


      SECTOR ARRVL
C         THIS SECTOR IS CONCERNED WITH ARRIVALS OF SHIPS
14    T.ARRIVE EQ 0
C         WHICH CAN ONLY OCCUR WHEN ONE IS DUE
      FIND S OCEAN FIRST &15
      S FROM OCEAN INTO HARBOUR
      T.S=0
C         FIND THE FIRST SHIP IN THE OCEAN MOVE IT TO THE
C         HARBOUR AND ZERO ITS TIME CELL
      T.ARRIVE=NEGEXP(7)
C         SAMPLE THE TIME TO THE NEXT ARRIVAL
      UNIFORM(SYSTEMSTREAM) GT 0.75 &13
      S INTO LARGE
      TLARGE+1
      GOTO 14
13    S INTO SMALL
      TSMALL+1
      GOTO 14
C         A QUARTER OF THE SHIPS ARE LARGE, OTHERS SMALL.
C         GO BACK TO START OF SECTOR IN CASE NEGEXP HAS
C         GIVEN A ZERO SAMPLE
15    WRITE(6,100) T.FINISH,CLOCK
100   LINGEN
```

```
1  NOT ENOUGH SHIPS IN MODEL - SIMULATION TERMINATED
2  TIME LEFT ***** TIME ELAPSED *****
  T.FINISH = 0
C     IF A SHIP IS NOT FOUND IN OCEAN, WRITE MESSAGE
C     AND SET T.FINISH SO THAT SIMULATION CEASES IN
C     SECTOR ENDNG.
GOTO ENDNG.
END


      SECTOR BTHL
C          THIS SECTOR IS CONCERNED WITH BERTHING LARGE SHIPS
      TIDE EQ 2
      FIND B FREE ANY
      FIND S HARBOUR FIRST
         S IN LARGE
C          THE TIDE MUST BE FULL, THERE MUST BE A FREE BERTH
C          AND A LARGE SHIP WAITING IN THE HARBOUR
      ENTER -T.S,LARGEQ
C          WHEN THE SHIP ENTERED THE HARBOUR ITS TIME CELL
C          WAS SET TO ZERO.  SINCE THEN IT HAS BEEN REDUCED
C          AT EACH TIME ADVANCE AND SO -T.S IS THE WAITING-
C          TIME OF THE SHIP.  THIS IS RECORDED IN THE
C          HISTOGRAM
      B FROM FREE INTO FULL
      S FROM HARBOUR INTO SHIPIN(B)
C          THE BERTH IS NOW FULL AND THE SHIP MOVES FROM THE
C          HARBOUR INTO THE BERTH
      ENTER -T.B,IDLE
C          JUST AS -T.S SHOWED THE SHIPS WAITING TIME SO -T.B
C          SHOWS THE BERTH IDLE TIME
      T.S=SAMPLE(UNLOD)
C          SAMPLE AN UNLOADING TIME FOR THE SHIP
      RECYCLE
C          CAUSE ANOTHER PASS THROUGH THE SECTORS (BECAUSE
C          MORE THAN ONE SHIP MIGHT BERTH AT THE SAME TIME)
      END


      SECTOR BTHS
C          THIS SECTOR IS CONCERNED WITH BERTHING SMALL SHIPS
C          AND IS SIMILAR TO THE PREVIOUS ONE
      TIDE GE 1
      FIND S HARBOUR FIRST
        S IN SMALL
      FIND B PART ANY  &20
C          THE SHIP IS MOVED TO A PARTLY FULL BERTH IF THERE
C          IS ONE
      SHIPIN(B) EQ 2  &30
      B FROM PART INTO FULL
C          IF THE BERTH ALREADY HAS TWO SHIPS IN IT, IT NOW
C          BECOMES FULL
      GOTO 30
```

```
20    FIND B FREE ANY
      B FROM FREE INTO PART
C         IF NO PARTLY FULL BERTH WAS FOUND, SEEK A FREE
C         BERTH WHICH NOW BECOMES PARTLY FULL
      ENTER -T.B,IDLE
C         RECORD IDLE TIME
30    ENTER -T.S,SMALLQ
      T.S=DEVIATE(5.0,20.0)
      S FROM HARBOUR INTO SHIPIN(B)
      RECYCLE
C         AS IN BERTHING OF LARGE SHIPS
      END


      SECTOR DBTH
C         THIS SECTOR IS CONCERNED WITH DEBERTHING
      TIDE NE 0
C         THE TIDE CANNOT BE OUT
      FOR X = 1,BERTH
C         DEAL WITH EACH BERTH SEPARATELY IN TURN
20       FOR S SHIPIN(X) FIRST      &15
          T.S LE 0
          CHAIN
            S IN SMALL
            OR S IN LARGE
            TIDE EQ 2
          DUMMY*
C         FIND A SHIP IN THE BERTH WHICH IS READY TO LEAVE
C         (TIME CELL HAS BEEN REDUCED TO ZERO OR BEYOND BY
C         TIME ADVANCE) AND WHICH CAN DO SO AT THE PRESENT
C         STATE OF THE TIDE.  IF NONE - GO ON TO TRY THE
C         NEXT BERTH
      RECYCLE
C         SET RECYCLE SWITCH TO TRY SECTORS AGAIN BEFORE
C         TIME ADVANCE (IN PARTICULAR BERTHING SECTORS
C         MAY NOW SUCCEED)
      S FROM SHIPIN(X) INTO OCEAN
      S IN LARGE    &16
      S FROM LARGE
      T.BERTH.X=0
      BERTH.X FROM FULL INTO FREE
      GOTO 15
C         IF SHIP LEAVING IS LARGE BERTH IS NOW FREE.
C         ZERO ITS TIME CELL SO THAT IDLE TIME CAN BE
C         COMPUTED LATER.  THEN GO TO NEXT BERTH.
16    S FROM SMALL
      SHIPIN(X) EQ 0    &17
      BERTH.X FROM PART INTO FREE
      T.BERTH.X=0
      GOTO 15
```

*DUMMY is a statement that does nothing when executed.

```
C          SIMILARLY IF SHIP LEAVING IS SMALL AND NOW THERE
C          ARE NONE LEFT IN THE BERTH.
17    SHIPIN(X) EQ 2    &20
      BERTH.X FROM FULL INTO PART
      GOTO 20
C          IF SMALL SHIP IS LEAVING AND BERTH WAS PREVIOUSLY
C          FULL, RECORD FACT THAT IT IS NOW ONLY PARTLY FULL
C          IN EITHER CASE GO BACK TO SEE IF ANY MORE SHIPS
C          ARE READY TO LEAVE THIS SAME BERTH
15    DUMMY
      DUMMY
      END



      SECTOR ENDNG
C          THIS SECTOR IS CONCERNED WITH OUTPUT OF RESULTS
      T.FINISH EQ 0
C          WHICH IS TO BE DONE AFTER TIME HAS BEEN ADVANCED
C          SO THAT T.FINISH HAS BECOME ZERO
      WRITE(6,100)
100   LINGEN SKIP PAGE
    1    PORT SIMULATION RESULTS
      WRITE(6,101)
101   LINGEN SKIP 2
    1    TOTAL    LARGE    SMALL
      WRITE(6,102) (TLARGE+TSMALL),TLARGE,TSMALL
102   LINGEN SKIP 1
    1    *****    *****    *****    SHIPS ENTERED HARBOUR
      J=0
      FOR S HARBOUR
       S IN LARGE    &10
       J+1
10     DUMMY
      K=HARBOUR-J
      WRITE(6,103)(J+K),J,K
103   LINGEN SKIP 1
    1    *****    *****    *****    SHIPS LEFT IN HARBOUR
      L=LARGE-J
      M=SMALL-K
      WRITE(6,104)(L+M),L,M
104   LINGEN SKIP 1
    1    *****    *****    *****    SHIPS STILL IN BERTHS
      TLARGE-J
      TSMALL-K
C          CALCULATE NUMBERS OF SHIPS THAT HAVE LEFT HARBOUR
      WRITE(6,100)
      WRITE(6,200)
200   LINGEN SKIP 2
    1    CELL RANGE    LARGEQ    SMALLQ    IDLETIME
      Y=0
      J=0
      K=0
```

```
        FOR X=1,25
             WRITE(6,300) Y,(Y+4),LARGEQ(X),SMALLQ(X),IDLE(X)
             Y+5
             J+LARGEQ(X)
             K+SMALLQ(X)
             DUMMY
        LONGL=TLARGE-J
        LONGS=TSMALL-K
C            TLARGE NOW HAS TOTAL NUMBER OF LARGE SHIPS WHICH
C            HAVE LEFT THE HARBOUR.  J HOLDS THE TOTAL NUMBER
C            OF ENTRIES IN THE HISTOGRAM.  THEREFORE TLARGE - J
C            IS THE NUMBER OF SHIPS WHOSE WAITING TIMES WERE
C            OUTSIDE THE RANGE OF THE HISTOGRAM
        WRITE(6,400) LONGL,LONGS
        STOP
300     LINGEN SKIP 1
    1      *** TO ***     *****     *****     *****
400     LINGEN SKIP 1
    1      OVER           *****     *****
        END
```

Description of the Program

The comments embedded in the program document its micro behavior rather well. What is not obvious from the program is its macro behavior, i.e., how activities are controlled, initiated, and performed.

The CONTROL segment of the program has four tasks: it defines global variables, it defines functions, it defines global FORMAT and LINGEN statements, and it specifies activities through the sector list. Of these, only the last is important to us now. Note that the CONTROL segment of CSL is similar to the SIMSCRIPT II preamble.

A short discussion of how time is represented within CSL models and how simulation is carried out should clarify the operations of the program.

Interactions in a real system are dependent on time and the system moves through time. It is therefore necessary to have some means of representing time in a simulation program. Time values are held in variables called T-cells. T-cells may arise in two ways; they are either defined as integer cells or arrays, or as cells attached to their names with "T". For example, if the class of ships is defined thus:

<u>CLASS</u> <u>TIME</u> <u>SHIP</u>.100

then this serves to define entity names as above, and also 100 T-cells addressed as T.SHIP.1,..., T.SHIP.100. An array of integer T-cells could be defined as

<u>TIME</u> BREAKDOWNS (10)

T-cells have all the properties of other integer cells and may participate normally in arithmetic and tests. Their time-advancing properties are additional.

Time advancement is performed in a repeated two-stage process as follows. Stage 1 scans all T-cells to find the smallest positive nonzero value in any cell. This is regarded as the time of the next event, or the time at which an event is next able to arise in the system. The program is now advanced to this position in time by subtracting this value from all T-cells. This completes stage 1.

In stage 2 the program itself is entered. The user must specify his program as a series of individual routines called activities, and phase 2 consists of an attempt to obey each of the activities in turn. Each activity describes the rules relating to the performance of one kind of activity in the system; for example, that of unberthing a ship.

The program statements in an activity normally begin with a series of tests to find out whether the activity can be initiated; these may be tests on T-cells to see whether, for instance, any ships are due to leave a berth. Following the opening tests are the statements that actually carry out the work of the activity, e.g., arithmetic and set-manipulation statements.

The actual question of division of the program into activities is governed by individual programming style. The activities must clearly cover all possible courses of action available in the system, but this is not the whole story. For instance, in the example program the berthings of large and small ships are handled as separate activities. The orders in these are largely duplicate; should they therefore be combined in one? This is purely a question of taste and as such is unresolvable on logical grounds.

The structure of a CSL test can now be more fully explained. The most frequent use of a test is at the start of an activity; under these circumstances, if the test fails, it may be assumed that the activity cannot be carried out.

For this reason, the customary operation of computer test orders has been changed; a test failure leads to transfer of control, usually to the next activity, whereas in the case of success the next statement is obeyed. To provide more detailed control of flow a statement label may be specified; e.g.,

```
DATA (10) EQ 4 & 87.
```

In event of failure, control goes to the statement labeled* 87.

The second phase, it will be noticed, consists of an attempt to obey all the activities specified in the system. Apparently, this involves much redundant effort, as at most points in time one or two activities only are likely to be entered successfully and the rest

*An & before a number indicates that it is a statement label.

will be abandoned after a test or two; but a closer analysis shows that computing time to carry out work of this kind must be expended in any simulation programming system, whether it is carried out under direct control of the programmer or not. It seems, therefore, most useful to make the necessary testing explicit and under the user's control.

When all the activities have been entered, the normal procedure is that a return to phase 1 takes place and time is further advanced. This procedure is not in itself sufficient; activities are interlinked and the completion of one activity may enable the initiation of another. For example, the unberthing of one ship will free a berth that another may use. The user can control this in two ways: first, by careful choice of the order in which activities are specified, and second, by the use of a special recycling device to cause further attempts to obey the activities to be made.

GPSS/360: A TRANSACTION-FLOW LANGUAGE

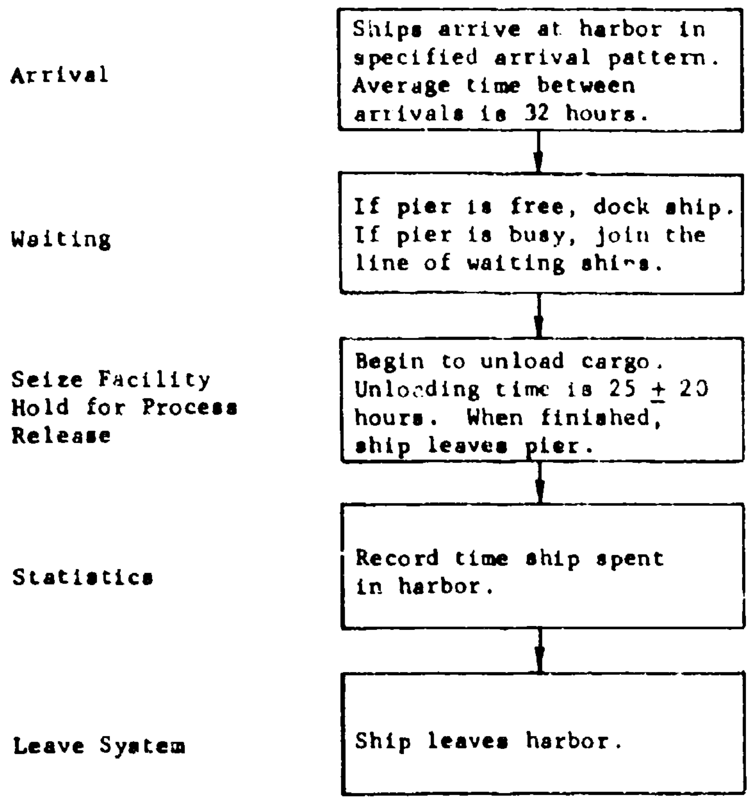
A simple harbor model is used to illustrate GPSS. While not identical to the CSL model, the match is close enough to provide some feel for how the languages represent similar systems. The example is taken from [24].*

The Model

Ships arrive at a small port with a known arrival pattern. While in port, the ships unload some of their cargo, taking a certain amount of time, and then proceed on their voyages. There being only one pier, a ship must wait if it arrives while another is unloading. If several ships are waiting, the one that arrived first will be unloaded first. Of interest here is the total amount of time a ship will spend in port, including the time spent waiting for the pier to become available.

*Reprinted by permission from (H20-0304-1 General Purpose Simulation System/360 Introductory User's Manual), © (1967), and (H20-0186-1 General Purpose Simulation System/360 Application Description), © (1966), by International Business Machines Corporation.

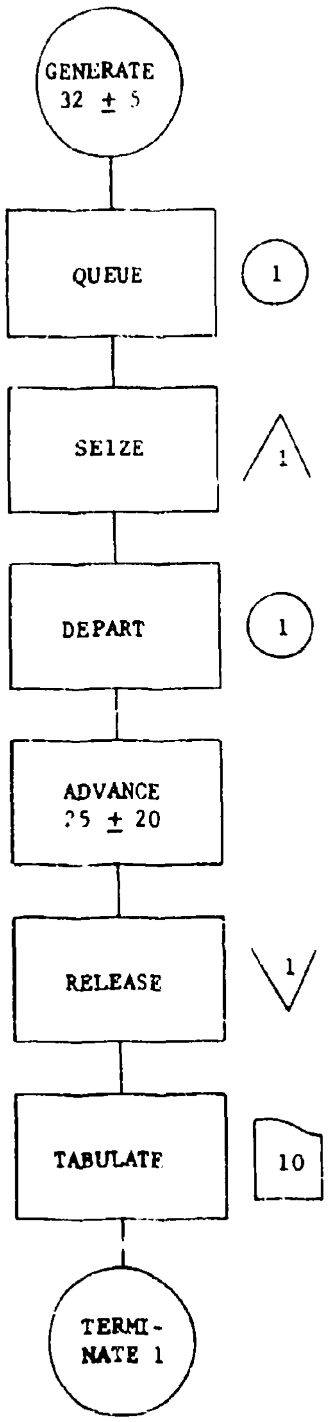
The gross behavior of the system can be pictured quite simply as:


Arrival
Ships arrive at harbor in specified arrival pattern. Average time between arrivals is 32 hours.
Waiting
If pier is free, dock ship. If pier is busy, join the line of waiting ships.
Seize Facility
Hold for Process
Release
Begin to unload cargo. Unloading time is 25 ± 20 hours. When finished, ship leaves pier.
Statistics
Record time ship spent in harbor.
Leave System
Ship leaves harbor.

<u>The Program</u>

Unlike most SPLs, GPSS has two representations, a flowchart and a coding form language. The flowchart model of the simple harbor system is shown in Fig. 10.

The coding form, or statement language model, is shown in Fig. 11. There is a direct correspondence between its statements and the flowchart symbols of Fig. 10.



Generate transactions (ships) at an average rate of one every 32 time units (hours).

Queue up transaction (ship) in queue 1, if facility 1 (pier) is busy.

Seize facility 1(pier) if it is free or, when it becomes free, make it busy.

Depart from queue 1, since transaction (ship) is no longer waiting for facility 1 (pier).

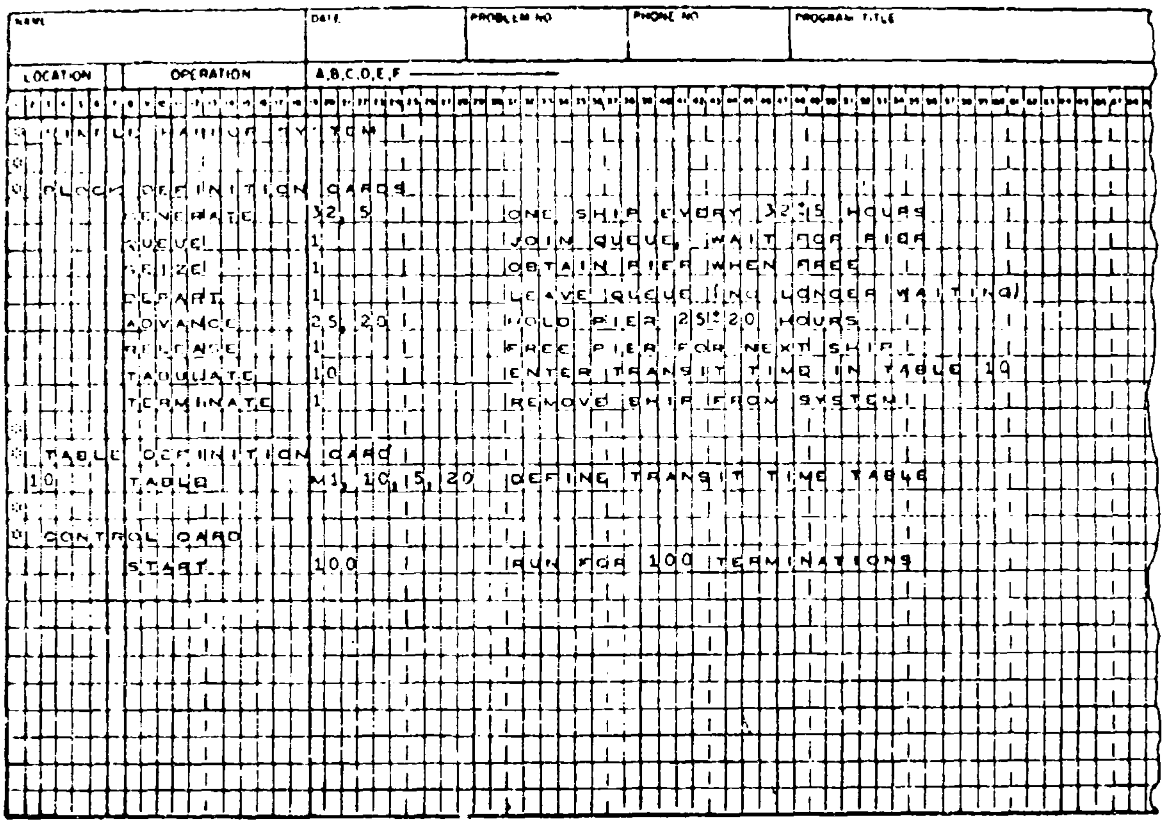
Advance time while this transaction is delayed (ship unloaded) for 25 ± 20 time units (hours).

Release facility 1 (pier), making it free.

Tabulate in Table 10 the total time spent by transaction (time ship was in harbor).

Terminate transaction (ship leaves harbor).

Fig. 10 -- GPSS flowchart for the simple harbor system



NAME | DATE | PROBLEM NO | PHONE NO | PROGRAM TITLE

LOCATION | OPERATION | A,B,C,D,E,F

```
*  SIMPLE HARBOR SYSTEM
*
*  BLOCK DEFINITION CARDS
          GENERATE    32,5         ONE SHIP EVERY 32±5 HOURS
          QUEUE       1            JOIN QUEUE, WAIT FOR PIER
          SEIZE       1            OBTAIN PIER WHEN FREE
          DEPART      1            LEAVE QUEUE (NO LONGER WAITING)
          ADVANCE     25,20        HOLD PIER 25±20 HOURS
          RELEASE     1            FREE PIER FOR NEXT SHIP
          TABULATE    10           ENTER TRANSIT TIME IN TABLE 10
          TERMINATE   1            REMOVE SHIP FROM SYSTEM
*
*  TABLE DEFINITION CARD
 10       TABLE       M1,10,15,20  DEFINE TRANSIT TIME TABLE
*
*  CONTROL CARD
          START       100          RUN FOR 100 TERMINATIONS
```

Fig. 11 -- GPSS Harbor Model

Description of the Program

The dynamic entities in GPSS are called "transactions." These represent the units of traffic, such as ships in this example. They are "created" and "destroyed" as required during the simulation run, and can be thought of as moving through the system causing actions to occur. Associated with each transaction are a number of parameters, to which the user can assign values to represent characteristics of the transaction. For example, a transaction representing a ship might carry the amount of cargo it is to unload in a parameter. This number could then be used in the simulator logic to determine how long the unloading operation would take. Transactions can be related to one another by placing them in groups that can be searched, scanned, and modified.

Entities of the second class represent elements of system equipment that are acted upon by transactions. These include facilities, stores, and logic switches. A facility can handle only one transaction at a time, and could represent the pier in the example given. It represents a potential bottleneck. A store can handle several transactions concurrently, and could be used to represent a parking lot or a typing pool. A logic switch is a two-state indicator that can be set by one transaction to modify the flow of other transactions. It could model a traffic light or the "next window" sign of a bank teller.

In order to measure system behavior, two types of statistical entities are defined: queues and tables. Each queue maintains a list of transactions delayed at one or more points in the system, and keeps a record of the average number of transactions delayed and the length of these delays. A table may be used to collect any sort of frequency distribution desired. These two entities provide a major portion of the GPSS output.

The operational entities, called "blocks," constitute the fourth and final class. Like the blocks of a diagram, they provide the logic of a system, instructing the transactions where to go and what to do next. These blocks, in conjunction with the other three classes of entities identified above, constitute the language of GPSS.

To provide input for the simulation, control and definition cards are prepared from a flowchart of the system. This constitutes the model in GPSS language. Once the system model is loaded, the GPSS program generates and moves transactions from block to block according to timing information and logical rules incorporated in the blocks themselves. Each movement is designated to occur at some particular point in time. The program automatically maintains a record of these times, and executes the movements in their correct time sequence. When actions cannot be performed at the originally scheduled time -- for example, when a required facility is already in use -- processing temporarily ceases for that transaction. The program automatically maintains a status of the condition causing the delay, and as soon as it changes, the transaction is activated again.

SUMMARY

Four different SPLs have been presented to give the reader a helpful glance at different concepts, features, and language styles in use today. Except for SIMULA, the examples illustrate the latest releases of the languages.* SIMULA 67 has been discussed in public [16], but since no written material was available at the time this Memorandum was written, an example of it is not included.

As these examples are small and simple, they do not illustrate all, or even necessarily the best, features of all four languages. Some languages fare better than others in these short destinations. The reader should bear in mind that the author's intent has not been language instruction, but a broad-based review. No language selections should be based on the details of this section alone. Still, it will be a useful exercise for the reader to look back and see how the different languages handle similar operations such as time control, entity generation, random sampling, and set manipulation.

---

*To be fair, it must be stated that of the four examples presented, the one describing GPSS/360 is the least representative of the power of its full language. The complete GPSS/360 language contains 14 entity and 43 block types.

V. CURRENT SPL RESEARCH

SPL research is currently going on in nonprofit corporations, universities, research organizations of computer manufacturers and computer software companies, industrial research organizations, and some military staff groups. With only a little simplification, one can say that this research falls in either of two general categories: the development of new simulation concepts, and the development of improved simulation systems. The two are quite different, yet, since all SPL projects seem to combine some elements of each, few people feel they are doing strictly one or the other. Researchers developing new concepts make advances in operating systems; experimenters developing new operating systems find new concepts arising from their work. While few projects can qualify as either pure concept development or pure operating system design, this distinction is made in the following discussion.

RESEARCH ON SIMULATION CONCEPTS

The 1967 IFIPS Working Conference on Simulation Languages [8] produced several important papers on SPL design.* Several of these described modifications to existing languages, others described new languages based on refinements of existing concepts. The spirit of the conference, however, was evolution rather than revolution.

Despite the fact that many SPLs are in use today, there have only been a few instances in which a language introduced concepts significantly different from its predecessors. The languages best known for introducing new simulation concepts are: GSP, GPSS, SIMSCRIPT, SOL, SIMULA, and SIMPAC.

So far as we know, no completely new simulation concepts are being developed today. Most language research is aimed at unifying existing language concepts (NSS [50] is an integration of SIMSCRIPT and SIMULA with some original ideas added**), extending an accepted

*IFIPS is the International Federation of Information Processing Societies.

**The most notable of these is a sophisticated version of the WAIT UNTIL command of SOL.

language (SIMSCRIPT II extends SIMSCRIPT as SIMULA 67 extends SIMULA), or writing a compiler for an existing language in a widely used procedural programming language (SPL [52] is being written in PL/I and is derived from SIMULA and SOL). A good deal of work being done throughout the programming community on data-base concepts is finding its way into simulation languages (e.g., SIMULA 67 and SIMSCRIPT II) and general-purpose programming languages (reference variable extensions to PL/I and ALGOL 68). Many concepts exploited in SPLs for some time are at present being integrated into COBOL.

There will always be room for this kind of research. Programming being what it is, it has no theoretical limit and there will always be opportunities for improvements, refinements, and extensions. Until a standard simulation programming language evolves, if that day ever comes, people will be rewriting SPLs in new languages, discovering new ways to do old things better, and making evolutionary changes in concepts and implementations.

A fertile field for SPL research is language concepts. One area in particular that has hardly been touched is the integration of discrete-event and continuous-time simulation. Today, continuous-time simulation is conducted on both analog and digital computers, with a trend toward increased use on digital and hybrid machines [6]. The languages used for simulating continuous systems on digital computers differ greatly from the SPLs we have been discussing. There is almost no relationship between discrete-event and continuous-time SPLs. This is a sad and hopefully short-term condition that will be alleviated when more research effort is expended on language integration.

A second promising area is the synthesis of modeling languages with procedures for performing statistical experiments and analyzing their results. While some work has been published on efficient statistical analysis and experimentation techniques [20], [49], no simulation programming languages presently contain such procedures. In part, this is because little research has been done on identifying and developing statistical procedures adapted especially to simulation studies. As this area receives more attention, however, both from researchers and practitioners, language designers will begin to consider experimentation and analyses as well as modeling and true "simulation systems" will be developed. Statisticians, language designers, simulation analysts and computer programmers will contribute jointly to such efforts.

To speak of evolution in simulation concepts and attempt to predict the future from the past, one can be guided by these facts:

Activity- and event-oriented SPLs emerged at roughly the same time. GSP, CSL, GPSS, and SIMSCRIPT were developed more or less in parallel.

Process-oriented SPLs came later. SOL and SIMULA evolved from the above languages and ALGOL.

Current research is attempting to unify and extend the activity-, event-, and process-orientations [8].

Interest in the statistical analyses and interpretation of simulation results seems to be growing more rapidly each year.

It seems a good bet that research in simulation modeling concepts will continue for some time. A great many modeling techniques still seem forced and artificial; there is much room for improvement in programming techniques. Topics that are well identified as needing research attention are: decision specification -- decision tables have not been used in SPLs; simultaneity -- parallel interactions are currently difficult to account for; synchronous and asynchronous behavior specification -- richer vocabularies for synchronizing system processes and for executing activities asynchronously are required; data-base definition -- we are still far from being able to specify complex state descriptions simply and elegantly; data-base management -- efficient ways of partitioning data bases and unifying them without complication remain to be worked out. For some solutions to these problems, see [4], [15], and [47].

RESEARCH ON OPERATING SYSTEMS AND MECHANISMS

As simulation is an experimental technique, people are always interested in making simulation programs easier to use. The standard way of performing a simulation experiment today is to make a series of experimental runs at different system parameter settings by submitting a set of programs and data decks for batch processing. While this procedure does the job, it has several serious defects: 1) program development and debugging is slow and painful; 2) more program runs than necessary are usually made because of the rigidity of a scheme requiring that a program be run completely before any information about its behavior can be obtained and used; and 3) it is difficult to get a feel for system dynamics by looking at a sequence of after-the-fact system state snapshots. Systems are being designed today that assist in each of these areas. These systems use interactive languages, time-sharing, and graphics.

Interactive Languages

The best-known interactive SPL is OPS-3 [27]. This language was designed and implemented at M.I.T. and used successfully to demonstrate the feasibility of interactive modeling. Operating in a time-shared environment under the M.I.T. Compatible Time Sharing System (CTSS), OPS-3 provides a user with on-line, interactive communication between himself and a programming system. Using it, one can, in a single sitting, compose a program, test and modify it, expand and embellish it, and prepare it for subsequent production use. That interactive languages will become standard in the future is a fact established by OPS-3, JOSS,* BASIC, RUSH, and other interactive languages. Greenberger and Jones [28] have specified in great detail the features of an elegant interactive simulation system.

Time-sharing is not mandatory for interactive man-machine dialogue, as the above discussion might imply. At M.I.T., an SPL named SIMPLE is being designed and programmed to operate on an IBM 1130 computer [17]. On such a small machine, it is economically feasible to allow one person to have full use, even though the computer is inactive a great deal of the time. Time-sharing allows multiple users to fully utilize a computer, but it is not necessary for an interactive language.

In the future, widely used SPLs will undoubtedly have two modes of operation. They will be able to be used interactively to build

*JOSS is the trademark and service mark of The RAND Corporation for its computer program and services using that program.

models; for this either incremental compilation or interpretation will be used. They will also be capable of efficient code generation through optimizing compilation. This is necessary if large simulation studies requiring lengthy experimental runs are to be made economically.

Time-Sharing

Time-sharing enters into simulation in that it makes certain things possible, such as interactive modeling. During model construction and testing, when programmer-program interaction is important, time-sharing makes interaction economical. Time-sharing can also be useful during model utilization, when production runs are made. When it is necessary to study a model's behavior, rather than run it merely to derive steady-state statistics, time-sharing can offer substantial benefits; when a man can enter a program from a console, watch its performance, and either leave it alone, stop it permanently, or stop it temporarily to adjust some parameters and start again, a new plateau in the use of simulation will be reached. It will then become possible for man to enter into the exploratory and experimental process more completely and more efficiently than he can in today's batch environment. For this reason, substantial future research will be devoted to this area. The SIMPLE language mentioned above is, in fact, designed to do this. IBM's continuous-time simulation language, CSMP, operates in this mode today [12].

Graphics

A considerable amount of simulation-oriented graphics research is going on right now. At the Norden Division of United Aircraft, an IBM 2250 is used to modify source language GPSS programs and view their output in graphical form [54]. At The RAND Corporation, the Grail system and the RAND Tablet are used to construct GPSS programs on-line from hand-drawn flowcharts [30]. At M.I.T., the SIMPLE project is using graphics for man-computer interaction during modeling and experimentation. Papers have been written on the use of graphics in simulation modeling and the use of existing graphics packages for analyzing simulation-generated output [44].

There is little doubt that interactive modeling is carried out better with a CRT device than with a typewriter or line printer. When designing programs, flowcharts and other symbolic notation can be displayed. For analyzing performance data, graphs provide more insight than do lists of numbers. The growing use of line-printer plotting simulators testifies to the utility of graphic output.

Research in SPL graphics will take several directions in the future. Graphical input is an area of great interest that is just getting started. Graphical output is on a far firmer footing and has had some operational success. The difficult thing about incorporating graphics in an SPL is its ultimate dependence on graphic hardware and software that are not properly part of a simulation language. Statements in SPLs that perform graphical tasks will, for a long time, be computer-system dependent, and not independent concepts. Graphical research will also prosper for interactive modeling and program modification. The successes and benefits claimed to date virtually insure this.

VI. THE FUTURE OF SPLS

The preceding section has demonstrated that a great deal of research remains to be done. We are far from knowing all there is to know about simulation, both in concept and in practice. It will be a long time before we come to a point where we wish to standardize on a single language, and change from a dynamic era of research and development to one of slow evolution.

The greatest challenge today lies in unifying discrete-event and continuous-time simulation languages. Some think that this cannot be done. Some think it should not be done. Certainly, we know little about how to do it.*

The researcher faced with the selection of a research project in the simulation language area does not lack alternatives. There are still advances to be made in modeling concepts; our ways of modeling both static and dynamic structures are incomplete. And certainly, the fields of interactive compilers and/or interpreters and graphics are exciting ones and in the mainstream of modern programming research.

Today, the manager or programmer faced with the task of selecting an SPL does not always have a clear choice. His choice will probably be less clear in the future, as languages are drawing closer together on many issues, but remaining apart on others. It is our hope that this Memorandum will make some language selection choices easier and more objective, and will provide some direction to SPL research.

*A recent publication, D. A. Fahrland, "Combined Discrete-Event/Continuous Systems Simulation," SRC-68-16, Case Western Reserve University Systems Research Center, July 1968, may provide the needed impetus to initiate a constructive dialogue on this topic.

APPENDIX

CHECKLIST OF FEATURES FOR SPL EVALUATION

Feature	Comments
MODELING A SYSTEM'S STATIC STATE	Simulation programming languages must be able to: (1) define the classes of objects within a system, (2) adjust the number of these objects as conditions within the system vary, (3) define characteristics or properties that can both describe and differentiate objects of the same class, and declare numerical codes for them, (4) relate objects to one another and to their common environment.
MODELING SYSTEM DYNAMICS	The heart of every simulation program, and every SPL is a time control program. Its functions are always the same: to advance simulation time and to select for execution a program that performs a specified simulation activity. An SPL must contain such a program, statements that define events, activities or processes, and statements that organise these events, activities or processes.
STATISTICAL SAMPLING	Pseudorandom number generation Multiple random number streams Antithetic variates Sampling from empirical table look-up distributions Sampling from theoretical statistical distributions
DATA COLLECTION SPECIFICATION	The nicest thing one can say about a data collection specification is that it is unobtrusive. While data collection is necessary, statements that are written to obtain data but are not themselves part of a simulation model's logic, should not obscure the operations of a model in any way.
DATA ANALYSIS	Means Variances and Standard Deviations Frequency Distributions
DATA COLLECTION	Number of observations, maxima and minima for all variables Sums, and sums of squares for time-independent variables Time-weighted sums, and sums of squares for time-dependent variables Variable value histograms for time-independent variables Time-in-state histograms for time-dependent variables Time series plots over specified time intervals
DISPLAY FORMATS	Since the production of reports is the primary task of all programs, whether they are run for program checkout, for the display of computed results, or for the preparation of elaborate management reports and charts, a good SPL should contain statements adapted to all display situations. (1) Automatic output in standard format (2) Format-free output (3) Formatted output (4) Report generators
MONITORING AND DEBUGGING	Reporting execute-time errors by source statement related messages Displaying complete program flow status when an execute-time error occurs. This means displaying the entry points and relevant parameters for all function and subroutine calls in effect at the time of the error. Accessing control information between event executions. This allows event tracing during all or selected parts of a program, and use of control information in program diagnostic routines.
INITIALIZATION	As simulation is essentially the movement of a system model through simulated time by changing its state descriptions, it is important that an SPL provide a convenient mechanism for specifying initial states. This can be done either by: (a) a special initialization form; or (b) convenient data input statements and formats. One should also be able to save all the information about a program, including relevant data on the status of external storage devices and peripheral equipment and restore it on command.
OTHER	Program readability Execution efficiency Modeling efficiency Documentation for instruction for installation for maintenance

REFERENCES

1. Arden, B. W., An Introduction to Digital Computing, Addison-Wesley, Inc., Reading, Mass., 1963.

2. Bennett, R. [illegible]., et al., "SIMPAC User's Manual," System Development Corporation, TM-602/000/00, April 1962.

3. Blunden, G. P., and H. S. Krasnow, "The Process Concept as a Basis for Simulation Modeling," SIMULATION, Vol. 9, No. 2, August 1967.

4. Blunden, G. P., "On Implicit Interaction in Process Models," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

5. Braddock, D. M., C. R. Dowling, and K. Rochelson, "SIMTRAN--A Simulation Programming System for the IBM 7030," IBM SDD, Poughkeepsie, N.Y., July 1965.

6. Brennan, R. D., "Continuous System Modeling Programs: State-of-the-Art and Prospectus for Development," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

7. Buxton, J. N., and J. G. Laski, "Control and Simulation Language," The Computer Journal, Vol. 5, No. 3, 1962.

8. Buxton, J. N. (ed.), Proceedings of the IFIP Working Conference On Simulation Languages, The North-Holland Publishing Company, Amsterdam, 1968.

9. Clementson, A. T., "Extended Control and Simulation Language," The Computer Journal, Vol. 9, No. 3, November 1966.

10. Colker, A., et al., The Generation of Random Samples from Common Statistical Distributions, United States Steel Corporation, Applied Research Laboratory Report 25.17-016(1), November 1962.

11. IBM United Kingdom Limited Data Centre, CSL User's Manual, London, 1966.

12. IBM Corporation, Continuous System Modeling Program (CSMP/360), Application Description, H20-0240, 1967.

13. Dahl, O. J., and K. Nygaard, "The SIMULA Language," Report from the Norwegian Computing Center, May 1965.

14. Dahl, O. J., and K. Nygaard, "SIMULA--An ALGOL-Based Simulation Language," Communications of the ACM, Vol. 9, September 1966.

15. Dahl, O. J., and K. Nygaard, "Class and Subclass Declarations," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

16. Dahl, O. J., B. Myhrhaug, and K. Nygaard, "Some Features of the SIMULA 67 Language," Proceedings of the Second Conference on Applications of Simulation, New York, December 2-4, 1968.

17. Donovan, J. J., J. W. Alsop, and M. M. Jones, "A Graphical Facility for an Interactive Simulation System," Proceedings of IFIPS Congress, 1968.

18. Draft Report on the algorithmic language ALGOL 68, working paper of the IFIP Working Group on ALGOL, (WG.2.1), February 1968.

19. Famolari, E., "FORSIM IV User's Guide," SR-99, The Mitre Corporation, February 1964.

20. Fishman, G. S., and P. J. Kiviat, Digital Computer Simulation: Statistical Considerations, The RAND Corporation, RM-5387-PR, November 1967.

21. Fishman, G. S., and P. J. Kiviat, "The Analysis of Simulation-Generated Time Series," Mgmt. Sci., Vol. 13, No. 7, March 1967.

22. Galler, B. F., The Language of Computers, McGraw-Hill Book Company, Inc., New York, 1962.

23. IBM, General Purpose Simulation System/360 User's Manual, H20-0326-2, 1967.

24. IBM, General Purpose Simulation System/360, Application Description, H20-0186-1, 1966.

25. Ginsberg, A. S., H. M. Markowitz, and P. M. Oldfather, Programming by Questionnaire, The RAND Corporation, RM-4460-PR, April 1965.

26. Gordon, G., "A General Purpose Systems Simulator," IBM Systems Journal, Vol. 1, 1962.

27. Greenberger, M., et al., On-Line Computation and Simulation: The OPS-3 System, The M.I.T. Press, Cambridge, Massachusetts, 1965.

28. Greenberger, M., and M. Jones, "On-Line, Incremental Simulation," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

29. IBM, IBM Operating System/360, PL/1: Language Specifications, File No. S-360-29, Form C28-6571-1, IBM Corporation, 1967.

30. Haverty, J. P., Grail/GPSS, Graphic On-Line Modeling, The RAND Corporation, P-3838, January 1968.

31. Hills, P. R., "SIMON—A Computer Simulation Language in ALGOL," in S. H. Hollingdale (ed.), Digital Simulation in Operational Research, American Elsevier Publishing Co., New York, 1967.

32. Kalinichenko, L. A., "SLANG--Computer Description and Simulation-Oriented Experimental Programming Language," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo Norway.

33. Karr, H. W., H. Kleine, and H. M. Markowitz, "SIMSCRIPT I.5," Consolidated Analysis Centers, Inc., CACI 65-INT-1, Santa Monica, Calif., June 1965.

34. Kiviat, P. J., Digital Computer Simulation: Modeling Concepts, The RAND Corporation, RM-5378-PR, August 1967.

35. Kiviat, P. J., "Development of Discrete Digital Simulation Languages," SIMULATION, Vol. 8, No. 2, February 1967.

36. Kiviat, P. J., "Development of New Digital Simulation Languages," Journal of Industrial Engineering, Vol. 17, No. 11, November 1966.

37. Kiviat, P. J., "GASP--A General Activity Simulation Program," Project No. 90.17-019(2), Applied Research Laboratory, United States Steel Corporation, Monroeville, Pa., July 1963.

38. Kiviat, P. J., Introduction to the SIMSCRIPT II Programming Language, The RAND Corporation, P-3314, February 1966.

39. Kiviat, P. J., R. Villanueva, and H. M. Markowitz, The SIMSCRIPT II Programming Language, The RAND Corporation, R-460-PR, October 1968.

40 Kiviat, P. J., Simulation Language Report Generators, The RAND Corporation, P-3349, April 1966.

41. Knuth, D. C., and J. L. McNeley, "SOL--A Symbolic Language for General-Purpose System Simulation," IEEE Transactions on Electronic Computers, August 1964.

42. Krasnow, H. S., "Dynamic Representation in Discrete Interaction Simulation Languages," in S. H. Hollingdale (ed.), Digital Simulation in Operational Research, American Elsevier Publishing Co., New York, 1967.

43. Krasnow, H. S., and R. A. Merikallio, "The Past, Present and Future of Simulation Languages," Mgmt. Sci., Vol. 11, No. 2, November 1964.

44. Lackner, M. R., "Graphic Forms for Modeling and Simulation," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

45. Markowitz, H. M., H. W. Karr, and B. Hausner, SIMSCRIPT: A Simulation Programming Language, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1963.

46. McNeley, J. L., "Simulation Languages," SIMULATION, Vol. 9, No. 2, August 1967.

47. McNeley, J. L., "Compound Declarations," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

48. Systems Research Group, Inc., MILITRAN Programming Manual, Report ESD-TDR-64-320, June 1964.

49. Naylor, T. H., et al., Computer Simulation Techniques, John Wiley and Sons, New York, 1966.

50. Parente, R. J., "A Language for Dynamic System Description," IBM Advanced System Development Division Technical Report 17-180, 1966.

51. Parslow, R. D., "AS: An ALGOL Simulation Language," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

52. Petrone, L., "On a Simulation Language Completely Defined Onto the Programming Language PL/I," presented at the 1967 IFIP Working Conference on Simulation Languages, Oslo, Norway.

53. Pritsker, A.A.B., and P. J. Kiviat, Simulation with GASP II: A FORTRAN Based Simulation Language, Prentice Hall, Inc., Englewood Cliffs, N. J. (forthcoming).

54. Reitman, J., "GPSS/360 Norden, The Unbounded and Displayed GPSS," Proceedings of SHARE XXX, Houston, Texas, February 29, 1968.

55. Report to the CODASYL COBOL Committee, COBOL Extensions to Handle Data Bases, prepared by Data Base Task Group, January 1968.

56. IBM, Simulation Evaluation and Analysis Language (SEAL), System Reference Manual, January 17, 1968.

57. United States Steel Corporation, Simulation Language and Library (SILLY), Engineering and Scientific Computer Services, February 1968.

58. General Electric Company, SIMCOM User's Guide, Information Systems Operations, TR-65-2-149010, 1964.

59. UNIVAC, SIMULA Programmer's Reference Manual, UP-7556, 1967.

60. Teichroew, D., and J. F. Lubin, "Computer Simulation: Discussion of Techniques and Comparison of Languages," Communications of the ACM, Vol. 9, No. 4, October 1966.

61. Tocher, K. D., "Review of Simulation Languages," Operational Research Quarterly, Vol. 16, No. 2, June 1965.

62. Tocher, K. D., and D. A. Hopkins, "Handbook of the General Simulation Program, II," Report 118/ORD 10/TECH, United Steel Companies, Ltd, Sheffield, England, June 1964.

63. Tocher, K. D., The Art of Simulation, D. Van Nostrand Company, Inc., Princeton, N. J., 1963.

64. Weinert, Arla E., "A SIMSCRIPT-FORTRAN Case Study," Communications of the ACM, Vol. 10, No. 12, December 1967.

65. Williams, J.W.J., "The Elliott Simulator Package (ESP)," Computer Journal, Vol. 6, No. 4, January 1964.

66. Young, Keren, "A User's Experience with Three Simulation Languages (GPSS, SIMSCRIPT and SIMPAC)," System Development Corporation, TM-1755/000/00, 1963.

DOCUMENT CONTROL DATA

1. ORIGINATING ACTIVITY THE RAND CORPORATION	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP
3. REPORT TITLE DIGITAL COMPUTER SIMULATION: COMPUTER PROGRAMMING LANGUAGES	
4. AUTHOR(S) (Last name, first name, initial) Kiviat, Philip J.	

5. REPORT DATE January 1969	6a. TOTAL No. OF PAGES 110	6b. No. OF REFS. 66
7. CONTRACT OR GRANT No. F44620-67-C-0045	8. ORIGINATOR'S REPORT No. RM-5883-PR	

9a. AVAILABILITY/LIMITATION NOTICES DDC-1	9b. SPONSORING AGENCY United States Air Force Project RAND

10. ABSTRACT

A discussion of simulation languages, their characteristics, the reasons for using them, and their advantages and disadvantages relative to other kinds of programming languages. Simulation languages are shown to assist in the design of simulation models through their "world view," to expedite computer programming through their special purpose, high-level statements, and to encourage proper model analysis through their data collection, analysis, and reporting features. Ten particularly important simulation programming language features are identified: modeling a system's static state, modeling system dynamics, statistical sampling, data collection, analysis and display, monitoring and debugging, initialization and language usability. Examples of each of the four simulation languages, GPSS, SIMSCRIPT II, SIMULA, and CSL, are used to illustrate how these features are implemented in different languages. The future development of simulation programming languages is dependent on advances in the fields of computer languages, computer graphics, and time sharing. Some current research is noted, and outstanding research areas are identified.

11. KEY WORDS

Computer simulation
Computer programming languages
Ships
Sealift
Statistical methods and processes